Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Data Race Analyse in SKilL Bauhaus

Daniel Quack

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dr. rer. nat. Timm Felden
Beginn am:	2017-10-17
Beendet am:	2018-05-17

Inhaltsverzeichnis

Kurzfassung

Das Projekt Bauhaus besteht aus zahlreichen verketteten Werkzeugen für eine Vielzahl von Programmanalysen. Innerhalb der Werkzeugkette wird die proprietäre Zwischendarstellung, die Intermediate Language (IML), als Serialisierungssprache zwischen den Werkzeugen verwendet. In einer Vorgängerarbeit wurden die generierten IML-Implementierungen des Bauhaus-Projektes in das *SKilL*-Binärformat angepasst. Die quelloffene Serialization Killer Language (SKilL) wurde an der Universität Stuttgart entwickelt und besitzt gegenüber der IML zahlreiche Vorteile. Die vorliegende Arbeit integriert ein Werkzeug der Data Race Analyse in die neu entstandene *SKilL*-Implementierung des Bauhaus-Projektes. Neben der erfolgreichen Portierung werden Konvertierungsregeln für weitere Migrationsprojekte erarbeitet. Die Ergebnisse wurden mithilfe einer eigens entwickelten Testumgebung validiert.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die das Gelingen dieser Arbeit ermöglicht haben. Ein besonderer Dank gilt dabei meinem Betreuer Dr. Timm Felden für die außerordentlich gute Betreuung dieser Arbeit. Seine Hilfestellung, sein Engagement und die fachliche Kompetenz trugen maßgeblich zum Erfolg dieser Arbeit bei. Weiterhin bedanke ich mich bei meinem Prüfer Prof. Dr. Erhard Plödereder für die Möglichkeit diese Masterarbeit am Institut für Softwaretechnologie zu schreiben.

Danken möchte ich außerdem meinen Freunden und Kollegen für ihre motivierenden und inspirierenden Worte, Korrekturen und Anregungen bei der Fertigstellung dieser Arbeit. Insbesondere danke ich Christiane Hofmann, Philipp Diersing und Dr. Markus Becker für die konstruktive Kritik an diesem Dokument.

Abschließend möchte ich mich ganz besonders bei meinen Eltern Karl-Peter und Angelika sowie meiner Schwester Carina für die Unterstützung während des gesamten Studiums bedanken.

1. Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Migration der Bauhaus-Zwischendarstellung nach *SKilL*.

Bauhaus ist eine Werkzeugkette zur statischen Code-Analyse von C-Programmen. Die verschiedenen Analysewerkzeuge tauschen gewonnene Programminformationen über die von Bauhaus eigenentwickelte binäre Zwischensprache IML aus. Eine Veränderung der IML-Spezifikation kann verhindern, dass ältere Binärdateien nicht mehr von den Werkzeugen verarbeitet werden können. Um diese Limitierung aufzuheben wurde an der Universität Stuttgart die Serialization Killer Language (SKilL) entwickelt.

SKilL ist eine quelloffene Sprache und erlaubt ein Maximum an Auf– und Abwärtskompatibilität. [Felden, 2017, Kapitel 1] Auf diese Weise können Formatänderungen, welche zum Beispiel durch neue Analysen erforderlich sind, effizient umgesetzt werden. Weiterhin erlaubt *SKilL* die sprach– und plattformunabhängige Serialisierung von großen Datenmengen. Neben den erwünschten Kompatibilitätseigenschaften verspricht eine Migration folglich weitere Vorteile.

In einem Vorgängerprojekt wurden die generierten Zwischendarstellungskompenenten bereits in das *SKilL*-Binärformat überführt. Handgeschriebene Komponenten, welche jedoch nicht in der zentralen Spezifikation abgebildet sind, konnten noch nicht migriert werden. Werkzeuge, die diese benutzen, sind somit in der *SKilL*-Version des Bauhausprojekts nicht nutzbar.

Darauf aufbauend zeigt diese Arbeit die Migration eines Werkzeuges der Data Race Analyse, welches handgeschriebene Zwischendarstellungskomponenten verwendet. Nach dem Erläutern der Grundlagen in Kapitel 2 und dem Beschreiben der Ziele in Kapitel 3 werden die erforderlichen Migrationsschritte in Kapitel 4 aufgezeigt. Anschließend werden in Kapitel 5 die Fehlerbilder zur Laufzeit diskutiert. Die Validierung der Ergebnisse mithilfe der Testsuite in Kapitel 6 sowie der Ausblick in Kapitel 7 schließen die Arbeit ab.

2. Grundlagen

In diesem Kapitel werden die Grundlagen für das Projekt Bauhaus behandelt. Weiterhin wird die Serialisierungssprache *SKilL* vorgestellt.

2.1. Bauhaus

Das Projekt Bauhaus[1] ist ein Gemeinschaftsprojekt der Universität Stuttgart, der Universität Bremen und der Axivion GmbH.[2] Bauhaus ist eine Toolsuite zur Softwarequalitätsanalyse von *C*- und *C++*-Programmen. Mögliche Programmanalysen sind unter anderem Klonerkennung, MISRA Checks, statische Code-Analysen, Qualitätsmetriken oder auch Race Conditions.

Zentrales Element der Bauhausinfrastruktur ist die Zwischensprache IML, die die zu analysierenden Programme in Form eines Abstract Syntax Tree (AST) darstellt. Der AST beinhaltet die syntaktischen und semantischen Informationen der Quelltextdateien [Raza et al., 2006, Kapitel 3.2] und ist der Ausgangspunkt für alle folgenden Analysewerkzeuge. Die sehr feingranulare IML kann weiterhin in den Resource-Flow-Graph (RFG) überführt werden, welcher den Code in den Funktions- und Methodenrümpfen auslässt und nur globale Programmelemente und ihre Abhängigkeiten als Graph darstellt. [Koschke, 2008, Seite 4] Dem RFG wird im Rahmen dieser Arbeit keine Bedeutung beigemessen, sondern lediglich aus Gründen der Vollständigkeit erwähnt.

Die IML wird mit dem Bauhaus Frontend erzeugt. Hierzu verwendet man anstatt eines herkömmlichen Compilers das Werkzeug cafecc. Dieses liest eine .c-Datei ein und ruft wiederum das Werkzeug cafe++ auf, das die benötigten semantischen und syntaktischen Informationen aus der Quelltextdatei herausliest und den IML-Graphen als Binärdatei erzeugt. Anschließend bindet der Linker imlink alle vorhandenen IML-Module zu einem großen IML-Graphen zusammen. [Przytarski, 2016, Seite 9]

Die IML-Binärdatei wird nun von den verschiedenen Analysewerkzeugen eingelesen und die entsprechenden Resultate erzeugt.

Die Werkzeuge der Toolsuite sind überwiegend in Ada geschrieben. Es existieren jedoch auch Schnittstellen zu *C*- und *C++*-Programmen. Tabelle 1 gibt einen Überblick über die derzeitige Projektgröße.

2.1.1. Zwischensprache IML

Wie bereits in Kapitel 2.1 eingeleitet, enthält die IML alle Informationen, die für die weiterführenden Analysen benötigt werden. [Koschke, 2008, Seite 4]
Der IML-Graph baut sich aus hierarchischen Klassenobjekten auf, die jeweils ein bestimmtes Sprachkonstrukt repräsentieren. Auf diese Weise werden semantisch ähnliche Sprachkonstrukte zusammengefasst. So erben zum Beispiel die WHILE-Schleife und die

[1] http://www.iste.uni-stuttgart.de/ps/projekt-bauhaus
[2] https://www.axivion.com/de

Sprache	Dateien	LoC
Ada	8418	1904446
C	808	201590
C++	326	158908
C/C++ Header	976	108572
...	...	...
SUM:	11968	2677385

Tabelle 1: Bauhaus-Projektübersicht analysiert mit CLOC. Die Anzahl der Dateien und Codezeilen sind nach der Programmiersprache kategorisiert.

FOR-Schleife beide von der allgemeineren LOOP-Klasse. Analysen, die nicht zwischen den Besonderheiten der verschiedenen Schleifentypen unterscheiden müssen, können direkt auf der LOOP-Klasse agieren. Ebenso wird eine programmiersprachenübergreifende Abbildung realisiert. So sind zum Beispiel die FOR-Schleifen von Ada und C-Programmen mit zwei verschiedenen FOR-Schleifen-Klassen definiert. Auch hier können Analysen, die zum Beispiel nicht an sprachspezifischen Schleifeneigenschaften interessiert sind, direkt auf den Elternklassen agieren.[Raza et al., 2006, Kapitel 3.2]

Die Definition der Zwischensprache wird über die IML-Spezifikation, die ebenfalls als IML bezeichnet wird, umgesetzt. Aus der Spezifikation wird dann mithilfe des Werkzeugs imlgen die API für die Spezifikation generiert. Die so generierten Dateien können von den Analysewerkzeugen verwendet werden, um die Informationen des Graphen zu verwerten.

Der Begriff IML kann somit je nach Kontext für die Spezifikation, die daraus generierte API oder den Graphen bzw. die Binärdatei verwendet werden. [Przytarski, 2016, Kapitel 2.1]

2.2. SKilL

Neben dem Bauhaus-Projekt aus Kapitel 2.1 betreibt das Institut für Softwaretechnologie (ISTE) auch das *SKilL*[3] (*Serialization Killer Language*) Forschungsprojekt.

SKilL ist ebenfalls eine Serialisierungssprache zur Verwaltung von Datenstrukturen in einer Binärdatei. Wie bei IML wird die zu generierende API mit einer Spezifikation beschrieben. Die genaue Syntax der Spezifikation kann der Arbeit [Felden, 2017] entnommen werden. Auch hier programmieren die Entwickler gegen die API und kommen mit der eigentlichen Binärdatei nicht in Berührung. [Harth, 2014, Kapitel 1.3] Im Mittelpunkt von *SKilL* steht die weitreichende Änderungstoleranz, die es ermöglicht, Strukturänderungen besser auszugleichen und die Wartbarkeit des Systems zu erhöhen. Die Java-ähnliche modulare Spezifizierungssprache ist weiterhin leicht verständlich und ermöglicht somit einen guten Einstieg für Entwickler.

[3] http://www.iste.uni-stuttgart.de/ps/projekt-skill.html

Ein weiterer Vorteil ist die Plattform– und Programmiersprachenunabhängigkeit von *SKilL*. [Felden, 2017, Kapitel 1.1] Im Rahmen von diversen Forschungs– und Studentenarbeiten wurden beispielsweise Sprachanbindungen an die Sprachen Ada, C, C++, Java, Haskell und Scala erarbeitet. Zugehörige Evaluationen wurde auf linuxbasierten Betriebssystemen, Mac OS X sowie Windows durchgeführt. [Przytarski, 2016, Kapitel 2.2.3] Diese Flexibilität erlaubt es den Entwicklern in den von Ihnen bevorzugten Sprachen zu entwickeln.

3. Ziele

Die vorliegende Arbeit ist ein Teil des in Kapitel 2.2 beschriebenen *SKilL*-Forschungsprojektes. Im Kontext dieser Forschung sollen die theoretischen Kenntnisse in einem praxisnahen Szenario nachgewiesen werden. Als Anwendungsfall dient die Integration von *SKilL* in das Bauhaus-Projekt. Die ursprüngliche Bauhaus-Zwischensprache IML wird im Rahmen dieses Migrationsprojektes durch *SKilL* ersetzt. Folglich beziehen sich die Begriffe *Migration* und *Migrationsprojekt* im weiteren Verlauf der Arbeit immer auf die beschriebene Portierung der Zwischensprache. Weiterhin wird die ursprüngliche Bauhaus-Version als *IML-Version* bezeichnet. Die Zielversion der Migration ist die *SKilL-Version*.

3.1. Istzustand

Konkret schließt diese Arbeit an die Masterarbeit Nr. 108 [Przytarski, 2016] an. In diesem Vorgängerprojekt wurden die generierten IML-Implementierungen an das *SKilL*-Binärformat angepasst. Abbildung 3.1 gibt einen Überblick über das Vorgehen.

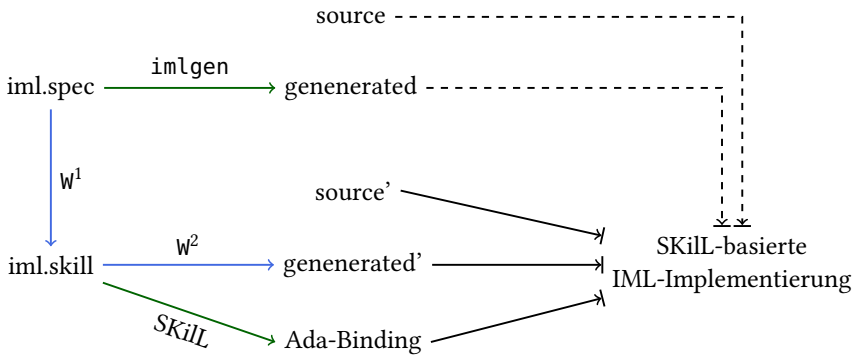

Abbildung 3.1: Istzustand des Bauhaus-Projekts, entnommen aus [Przytarski, 2016, Kapitel 3.2]

Der obere Teil der Grafik veranschaulicht die ursprüngliche Architektur des Bauhaus-Projektes. Aus der IML-Spezifikation *iml.spec* wird mithilfe des Werkzeuges imlgen die generierte API, mit welcher auf den IML-Graphen zugegriffen werden kann, erzeugt. Zusätzlich existieren handgeschriebene Komponenten (*source*), die die generierte API nutzen.

In der Masterarbeit Nr. 108 wurden zwei Werkzeuge specgen und codegen entwickelt, die die Portierung ermöglichen. Specgen, in der Grafik mit w^1 gekennzeichnet, generiert aus einer IML-Spezifikation die äquivalente *SKilL*-Spezifikation. Ausgehend von dieser kann mit dem Werkzeug *SKilL* das Ada-Binding erzeugt werden. In diesem sind zum Beispiel die in der Spezifikation angegebenen Datenstrukturen in Ada definiert. Mit dem zweiten entwickelten Werkzeug codegen (w^2) kann aus der *SKilL*-Spezifikation eine IML API (*generated'*) generiert werden. [Przytarski, 2016, Kapitel 5] Die in der Grafik dargestellte *source'* Komponente veranschaulicht handgeschriebene Komponenten, die

bereits zusammen mit dem Ada-Binding und den durch codegen generierten Dateien verwendet werden. Die beiden Werkzeuge specgen und codegen sind Teil des Werkzeuges imlgen4.

3.2. Sollzustand

Das Ziel dieser Masterarbeit ist die Migration der Werkzeuge zur Erkennung von Data Races in C-Programmen. Die vier betroffenen Werkzeuge, pta_tool, iml2cfg, thread_tool sowie raceq, verwenden Zwischendarstellungskomponenten, welche nicht in der zentralen Spezifikation abgebildet sind. Folglich konnten diese Analysewerkzeuge nicht im Rahmen der automatischen Migration der Vorgängerarbeit portiert werden. Konkret bedeutet dies, die nicht compilierbaren Zwischendarstellungskomponenten auf die von *SKilL* generierte Äquivalente umzustellen. Die Funktionsfähigkeit der migrierten Werkzeuge soll durch einen Vergleich mit der alten Implementierung gezeigt werden. Weiterhin soll ein Testwerkzeug implementiert werden, mithilfe dessen sich die Funktionsfähigkeit der Werkzeuge automatisch prüfen lässt.

3.2.1. Vorgehensweise

Der Aufruf der zu portierenden Werkzeuge erfolgt sukzessive. Das Programm pta_tool erweitert den eingelesen *SKilL*-Graph eines .c-Programms mit den für die Data Race Analyse benötigten Informationen. Das letzte Werkzeug dieser Kette, raceq, gibt Warnungen über mögliche Data Races aus. Der Informationsaustausch zwischen den Werkzeugen erfolgt über die *SKilL*-Binärdateien.

Infolgedessen muss die Portierung ebenfalls entlang der Aufrufreihenfolge der Analysewerkzeuge erfolgen, weshalb mit der Portierung des pta_tools begonnen wurde. Zu Beginn der pta_tool Migration zeichnete sich ein enormer Aufwand für die manuelle Integration der Werkzeuge ab. Die Werkzeuge besitzen zahlreiche Abhängigkeiten zu weiteren nicht funktionsfähigen Bauhauskomponenten, die in diesem Umfang nicht einkalkuliert waren. Eine detaillierte Beschreibung des Mehraufwandes des pta_tools wird in Kapitel 7.1.1 gegeben. Aus diesem Grund verlagerten sich auch die Ziele dieser Arbeit.

Im ersten Schritt soll die Machbarkeit der Portierung der handgeschriebenen Zwischendarstellungskomponenten am Beispiel des pta_tools gezeigt werden. Hierfür wird eine Beschreibung erstellt, wie verwendete Ada-Programmkonstrukte und Datenstrukturen der handgeschriebenen IML-Komponenten in die verfügbaren *SKilL*-Pendants konvertiert werden können. Die entworfenen Konvertierungsregeln sind unabhängig von den Data Race Werkzeugen und können auch bei allen übrigen Werkzeugen angewendet werden. Die Validierung erfolgt mithilfe der angedachten Referenzmessung gegen das IML pta_tool.

4. Migration

Dieses Kapitel enthält eine strukturierte Auflistung der Migrationsthemen, um eine handgeschriebenen IML-Komponenten in eine compilierfähige *SKilL*-Software zu migrieren. Mögliche Fehlerbilder werden aufgezeigt und die erarbeiteten Lösungsvorschläge dargestellt.

4.1. Auslagerung der serialisierten Datentypen

Die Idee von *SKilL* verlangt, dass alle Datenstrukturen, die über die Zwischendarstellung ausgetauscht werden können, in der *SKilL*-Spezifikation definiert sind. Aus der Spezifikationsdatei wird mithilfe eines Codegenerators das Ada-Binding generiert. Von den generierten Dateien ist insbesondere die Datei *siml.ads* für diese Arbeit relevant. Diese enthält die Ada-Typdefinitionen der in *SKilL* spezifizierten Datenstrukturen. Auf diese Weise können die spezifizierten Typen in den in Ada implementierten Werkzeugen verwendet werden. Folglich müssen die zu migrierenden Dateien auch ausschließlich die generierten Typen verwenden. Mit anderen Worten dürfen in den *.ads*-Dateien der Werkzeuge keine Typdefinition von serialisierbaren Datenstrukturen vorliegen. Dieses Kapitel beschreibt, wie dieser Austausch der Datenstrukturen zu erfolgen hat. Quellcode 1 zeigt exemplarisch die generierte Definition des Typs *C* im Ada-Binding *siml.ads*.

```
1 type C_T is new Skill.Types.Skill_Object with private;
2
3 type C is access all C_T;
4 type C_Dyn is access all C'Class
```

Quellcode 1: Exemplarische Definition des Typs *C* im Ada-Binding *siml.ads*

Der Postfix *_T* markiert den eigentlichen Typ im Ada-Binding. Je nach Bedarf und Kontext können die beiden Zeiger *C* und *C_Dyn* verwendet werden, um auf Variablen vom Typ *C* zuzugreifen. Weiterhin stellt *siml.ads* bzw. *siml.adb* Hilfsfunktionen (Quellcode 2) zur Verfügungen, mit welchen zwischen den verschiedenen Zeigertypen gewechselt werden kann.

```
1 function To_C (This : Skill.Types.Annotation) return C
2  with Inline, Pure_Function;
3
4 function Unchecked_Access (This : access C_T) return C;
5 pragma Inline (Unchecked_Access);
6
7 function Dynamic_C (This : access C_T) return C_Dyn;
8 pragma Inline (Dynamic_C);
```

Quellcode 2: Hilfsfunktionen zur Typkonvertierung im Ada-Binding

Im Rahmen der Migration werden folglich alle serialisierbaren Typdefinitionen in den .ads-Dateien der Werkzeuge ersetzt. Gemäß der Vorgehensweise in [Przytarski, 2016, Kapitel 5.1] werden Untertypen eingeführt, die auf das Ada-Binding verweisen. Die Definition eines Typs C in einer .ads-Datei wird durch die Untertypen in Quellcode 3 ausgetauscht.

```
1 use type Siml.C_T;
2
3 subtype C_Class is Siml.C_T;
4 subtype C is Siml.C_Dyn;
```

Quellcode 3: Die ausgelagerten Datenttypen werden durch neue Untertypen ersetzt.

Das Auslagern der Typdefinitionen verursacht bei der Migration zahlreiche Compilerfehler, welche in den nachfolgenden Kapiteln genauer beschrieben werden. Die häufigsten Probleme sind:

- Durch das Verwenden der *SKilL*-Basistypen geht die Namensäquivalenz zu den lokal genutzten Typen verloren. Das strenge Typsystem von Ada verursacht folglich eine Vielzahl an Compilerfehlern bei sämtlichen Operationen mit den neuen Untertypen: Siehe Kapitel 4.3.
- Die Ada-Präfixnotation ist nicht mehr möglich, da die Typdefinitionen nicht mehr sichtbar sind: Siehe Kapitel 4.4.
- Auf die Felder von *Records* kann nicht mehr mit der Punktnotation zugegriffen werden: Siehe Kapitel 4.5
- Abstrakte Funktionen können nicht mehr abstrakte Typen verwenden, da diese jetzt echte Untertypen sind: Siehe Kapitel 4.6.
- Viele ehemals vorhandenen Datenstrukturen sind in der *SKilL*-Version anders dargestellt. Zum Beispiel Arrays, Maps, Enums, Variant Records: Siehe Kapitel 4.7.

4.2. Allokation neuer SKilL-Objekte

Werden in der IML-Version Instanzen von ausgelagerten Datenstrukturen allokiert, so müssen diese aufgrund der Migration in den *SKilL*-Graph eingefügt werden.

Zur Lösung wird das Hilfspaket *SKilL-State* verwendet. Dieses verschafft Zugriff auf die Graphinstanz und ermöglicht das Hinzufügen von neuen *SKilL*-Objekten. In Quellcode 4 wird exemplarisch die Variable *foo* vom Typ T allokiert.

Je nach Struktur von T erwartet *make* noch entsprechende Übergabeparameter. Weitere Informationen zu dem Paket *SKilL-State* sind in [Przytarski, 2016, Kapitel 6.7] zu finden.

```
1 foo := Skill_State.Get_Constructor_Graph.T.Make;
```

Quellcode 4: Allokation einer serialisierbaren Variablen vom Typ T

4.3. Cast zu SKilL-Basistypen

Durch die Migration entstehen Typinkompatibilitäten aufgrund unterschiedlicher Basistypen. *SKilL* verwendet zur Repräsentation von *Integer* den *SKilL* Builtin-Typ *v64*. [Felden, 2017, Kapitel 4.1] *V64* ist ein *Integer* mit einer variablen Länge im serialisierten Zustand. Eine *v64*-Variable hat im Hauptspeicher immer eine Länge von 64 Bit, im *SKilL*-Zustand kann sie je nach Wertebereich ein bis neun Byte betragen. [Przytarski, 2016, Kapitel 2.2.1] Quellcode 5 zeigt die entsprechende Darstellung in Ada. Es ist erkennbar, dass sich die Untertypdefinition des variablen Typs *v64* tatsächlich nicht von der Definition des Integers mit fester Länge – *i64*– unterscheidet.

```
1 subtype i8 is Interfaces.Integer_8 range Interfaces.Integer_8'Range;
2 subtype i16 is Interfaces.Integer_16 range Interfaces.Integer_16'Range;
3 subtype i32 is Interfaces.Integer_32 range Interfaces.Integer_32'Range;
4 subtype i64 is Interfaces.Integer_64 range Interfaces.Integer_64'Range;
5 subtype v64 is Interfaces.Integer_64 range Interfaces.Integer_64'Range;
```

Quellcode 5: *SKilL*-Typdefinitionen in Ada in skill-types.ads

Alle in der *SKilL*-Spezifikation definierten Datentypen verwenden nun folglich ebenfalls den *v64* Typ bzw. den Ada Typ *Integer_64*. Die handgeschriebenen Dateien verwenden jedoch den vordefinierten Ada *Integer*-Typ oder von diesem abgeleitete Typen. Beim Ein– und Auslesen der im Ada-Binding definierten Typen innerhalb der handgeschriebenen Dateien liegen nun Typinkompatibilitäten vor. Zur Lösung existieren zwei Ansätze.

Die konsistente Lösung wäre die entsprechende Anpassung der Typdefinitionen in den betroffenen Dateien, sodass auch alle Unterprogramme mit *Integer_64* arbeiten. Dieser Ansatz ist jedoch schwer umsetzbar. Erste Anpassungen diesbezüglich zu Beginn dieser Arbeit führten zu einer enormen Anzahl an Folgefehlern. Die Änderungen hatten Auswirkungen auf eine Vielzahl von weiteren Quelldateien, die wiederum eine Anpassung in ihren Spezifikationsdateien (*.ads) erforderten. Eine Kettenreaktion von Compilerfehlern über das komplette Bauhaus-Projekt ist die Folge. Aufgrund dessen wurde zu diesem Zeitpunkt der Aufwand zur Fehlerbehebung mit diesem Ansatz als zu groß eingestuft. Auch lassen sich nicht alle *Integer*-Typen pauschal konvertieren. Je nach Kontext werden mit den Werten auch Indices von indizierbaren Datenstrukturen errechnet, welche weiterhin einen *Integer*-Typ als Index führen. In diesem Fall hätten indizierte Zugriffe nun eine Typinkompatibilität. Zu guter Letzt sollten auch Änderungen außerhalb der Werkzeuge für die Data Race Analyse, welche durch die beschriebene Fehlerkaskade möglich wären, vermieden werden.

Der wesentlich einfachere und direktere Ansatz sind Casts beim Übergang vom Ada-Binding zu den zu migrierenden Werkzeugen. Beim Lesen und Schreiben von in *SKilL* definierten Typen und Feldern, die einen *Integer*-Typ enthalten, erfolgt ein direkter Cast. Konkret wurden Funktionsparameter, Returnwerte und Typdefinitionen nicht verändert. Lokale Variablendefinitionen sind, je nach notwendiger Castanzahl, jedoch konvertiert worden. Auf diese Weise konnten Folgefehler im weiteren Programmfluss unterbunden werden. Der Vorteil dieser Methode ist die sichtbare Reduzierung der Fehleranzahl. Es werden keine Folgefehler ausgelöst und die internen Bauhaus-Schnittstellen bleiben diesbezüglich unberührt. Auch der bestehende Quellcode wird in Hinblick auf die IML-Implementierung nicht verändert, indem auf einmal anstatt mit 32 Bit Werten mit 64 Bit Werten gerechnet wird. Der offensichtliche Nachteil dieser Vorgehensweise ist eine Reihe von unnötigen Folgecasts bei Funktionsverkettungen. Zudem verringern die vielen Casts die Lesbarkeit des Codes.

Auch bei diesem Ansatz war der erforderliche Änderungsaufwand hoch. Das Einfügen von Casts lässt sich aber effizient mithilfe von Skripten automatisieren.

4.4. Auflösen der Ada-Präfixnotation

Das Auslagern der Typdefinitionen in das Ada-Binding und das Verwenden der Untertypen in den zu migrierenden Paketen verhindert die Verwendung der Ada-Präfixnotation.

Die Präfixnotation wurde in Ada 2005 eingeführt und ist eine elegante Möglichkeit, ausgehend von einem Objekt die dazugehörigen Unterprogramme aufzurufen. [ISO/IEC, 2012, Kapitel 4.1.3] Barnes [Barnes, 2014, 14.4] erklärt die Präfixnotation wie folgt. Eine Operation *OP* auf einem *tagged* Typ T, welcher im Paket P deklariert ist, kann wie folgt umgeschrieben werden:

```
P.Op (X, otherParams) -- Paket Notation
X.Op (otherParams) -- Präfixnotation
```

Vorausgesetzt X ist vom Typ T und ist der erste Parameter des Unterprogramms. Zusätzlich ist T ein *tagged* Typ und *OP* ist eine primitive oder klassenweite Operation auf T.

Das Auslagern der Typdefinition und das Einführen von Untertypen mit denselben Namen lässt alle Präfixnotationen mit *SKilL*-Typen scheitern. Als eine Folge müssen alle vorkommenden Präfixnotationen durch die explizite Paketnotation ersetzt werden. Analog zum Vorgehen in Kapitel 4.3 kann auch dieser Fehlertyp effizient mithilfe von Skripten behoben werden. Zu Inspirationszwecken, ist im Anhang C das verwendete Skript abgebildet.

4.5. Zugriff auf Felder

Der Zugriff auf die Felder von ausgelagerten *Records* erfolgt nun mithilfe von Getter– und Setter-Methoden. Für jedes in der *SKilL*-Spezifikation definierte Feld werden im Ada-Binding die korrespondierende Getter– und Setter-Methoden generiert. Die generierten Unterprogramme können mit den neuen Untertypen aus Quellcode 3 verwendet werden.

```
1 function Get_Id (This : not null access constant C_T'Class) return
      Standard.Skill.Types.V64;
2 procedure Set_Id (This : not null access C_T'Class; V : Standard.
      Skill.Types.V64);
```

Quellcode 6: Beispielhafte generierte Getter– und Setter-Methoden im Ada-Binding. Sie ermöglichen einen Zugriff auf das Feld *Id* des Typs *C* aus Quellcode 1.

Analog zu Kapitel 4.3 und Kapitel 4.4 können auch hier Skripte zur Fehlerbehebung eingesetzt werden.

4.6. Auflösen von abstrakten Funktionen und Prozeduren

Bei dem in Kapitel 4.1 beschriebenen Auslagern der Datentypen kommt es in dem Paket *DF_Pattern* zu einer weiteren Besonderheit. Das Paket definiert einen abstrakten *Record* und mehrere abstrakte Funktionen, die diesen *Record* als Parameter verwenden. Das Ersetzen des *Records* in einen ausgelagerten *SKilL*-Untertyp hindert den Compiler daran, abstrakte Funktionsaufrufe innerhalb dieses Paketes aufzulösen. Im Folgenden wird eine exemplarische Fehlermeldung für diesen Fall aufgezeigt.

```
error: cannot call abstract subprogram "..."
error: access to class-wide argument not allowed here
```

Um dieses Problem zu lösen, kann man nun die ehemalige abstrakte Variable *downcasten* und explizit die richtige Unterfunktion aufrufen. Um die dafür notwendigen Typprüfungen zur Laufzeit möglichst gering zu halten, wurde das Problem mit folgender Analyse deutlich vereinfacht.

Zunächst soll die Typhierarchie in Abbildung 4.1 betrachtet werden. Nach mündlicher Rücksprache mit Timm Felden wurde festgestellt, dass das Paket *Simple_DF_Pattern_Class* nicht mehr verwendet und benötigt wird. Daher wurde im Rahmen dieser Fehlerbehebung das Paket vollständig entfernt und der hierarchische Baum somit stark vereinfacht.

Weiterhin wurde mit Tabelle 2 eine Übersicht aller abstrakten Funktionen, die in *DF_Pattern_Class* deklariert sind, erstellt. Die beiden Spalten *PTA_DF_Pattern_Class* und *Thread_DF_Patterns* signalisieren, ob die angegebene Funktion im jeweiligen Paket implementiert ist. Es wird deutlich, dass ein Großteil der Funktionen nur in einem

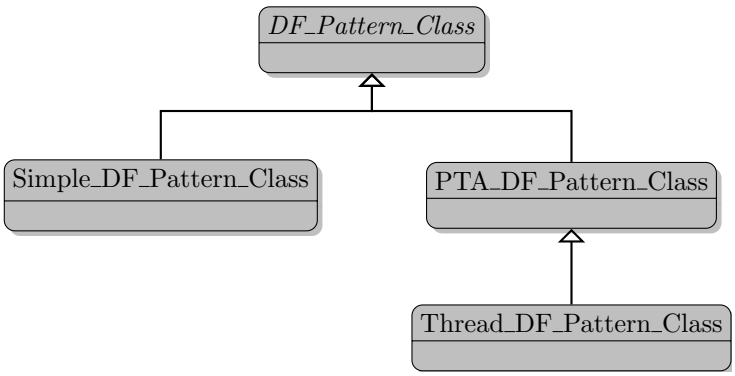

Abbildung 4.1: Typhierarchie der *DF_Pattern_Class* in der IML Version

Funktion	PTA _DF_Pattern	Thread _DF_Pattern	Eigenschaften
Get_Locators	Ja	Nein	–
Image	Ja	Nein	AK
Get_Object	Ja	Nein	–
Get_Must_Defs	Ja	Nein	TC
Get_May_Defs	Ja	Ja	AF + TC
Get_May_Uses	Ja	Ja	TC
Get_Caller_Patterns	Ja	Ja	TC
Get_Callee_Patterns	Ja	Ja	TC
Must_Map_To_Caller	Ja	Nein	AF
May_Map_To_Caller	Ja	Nein	AF
Must_Map_To_Callee	Ja	Nein	AF
May_Map_To_Callee	Ja	Nein	AF
Has_Visible_Out_Effect	Ja	Nein	TC
Compute_Must_Def_Set	Ja	Nein	AK + TC
Compute_May_Def_Set	Ja	Nein	TC
Compute_May_Use_Set	Ja	Nein	TC

Tabelle 2: Funktionsübersicht der abstrakten Funktionen und Prozeduren im Paket DF_Patterns. Die weiter unten erläuterten Eigenschaften sind wie folgt abgekürzt: AK (Auskommentiert), TC (Toter-Code) oder AF (Abstrakter Funktionsaufruf)

Paket vorhanden ist. Die Frage nach dem *Dispatching* erübrigt sich in diesen Fällen. Die Variable vom Typ *DF_Pattern_Class* wird in ein *PTA_DF_Pattern* konvertiert und die Funktion aus demselben Paket explizit aufgerufen.

Der zweite Fall, bei dem eine Implementierung in beiden Paketen vorhanden ist, verlangt stattdessen eine Laufzeitprüfung. Zu diesem Zweck werden die betroffenen abstrakten Funktionen und Prozeduren in *DF_Pattern_Class* in konkrete Unterprogramme umgewandelt. Der *Body* dieser Methoden enthält nun eine Typprüfung zur Laufzeit. Nach der Typidentifikation wird wieder entsprechend konvertiert und das richtige Unterprogramm explizit aufgerufen. Quellcode 7 zeigt die verwendete Typprüfung am

Beispiel der nun konkreten Funktion *Get_May_Defs* in *DF_Pattern_Class*.

```
1 name := Pattern.Dynamic_DF_Pattern.Skill_Name;
2 if name = Siml.Internal_Skill_Names.Threaddfpattern_Skill_Name then
3     return Thread_DF_Patterns.Get_May_Defs(...)
4 elsif name = Siml.Internal_Skill_Names.Ptadfpattern_Skill_Name then
5     return PTA_DF_Patterns.Get_May_Defs(...)
6 else
7     raise Program_Error with "Didn't find right subprogram for this
          type.";
8 end if;
```

Quellcode 7: Manuelles Dispatching zur Auflösung ehemaliger abstrakter Funktionsaufrufe in Pseudocode in dem Paket *DF_Pattern*

Tabelle 2 enthält zudem die vierte Spalte „Eigenschaften“. Diese Spalte enthält Zusatzinformationen, die im Rahmen dieser Analyse festgestellt worden sind. Die folgende Aufzählung definiert die eingeführten Abkürzungen.

- AK (Auskommentiert): Die Funktion ist in beiden Paketen implementiert, aber in dem mit Nein deklarierten Paket bereits in der IML-Version auskommentiert. Der auskommentierte Status wurde in dieser Arbeit nicht verändert.
- AF (Abstrakter Funktionsaufruf): Die Funktion wird im *DF_Pattern* Paket aufgerufen und verursachte somit den zu Beginn dieses Kapitels beschriebenen Compilerfehler. Je nach Szenario wurde eine der beiden beschriebenen Lösungsansätze angewendet.
- TC (Toter-Code): Bei den mit TC gekennzeichneten Funktionen handelt es sich nicht um 100 prozentigen Toten-Code. Es existieren durchaus vereinzelte Aufrufe zu diesen Methoden. Die aufrufenden Pakete sind zum jetzigen jedoch nicht compilierbar (zum Beispiel Slicing oder SSA-Analyse). Aus dem Programmfluss des pta_tools oder dem übrigen compilierbaren Bauhaus-Code sind die TC-Funktionen jedoch nicht erreichbar. Da durchaus die Möglichkeit besteht, dass sie im Rahmen weiterer Migrationen benötigt werden, beispielsweise bei einer Instandsetzung der SSA-Analyse, wurden die Funktionen nicht gelöscht, sondern lediglich mit einem entsprechenden Hinweis auskommentiert.

Toter-Code-Analyse (TCA) Die durchgeführte TCA ist ein Nebenprodukt der notwendigen Fehlerbehebung der beschriebenen Problematik dieses Kapitels. Eine statische Aufrufanalyse der in Tabelle 2 gelisteten Funktionen erbrachte folgende Erkenntnis. Alle mit TC markierten Funktionen besitzen zahlreiche verkettete Aufrufe untereinander, es gibt jedoch nur wenige Funktionsaufrufe von anderen Paketen. Diese externen Aufrufe erfolgen ausschließlich von nicht compilierbaren Einheiten, was die TC Unterprogramme unerreichbar macht.

Insgesamt wurden durch diese Analyse 652 Zeilen Code auskommentiert. Da diese Sachlage erst gegen Ende dieser Arbeit entdeckt wurde, sind alle auskommentierten Unterprogramme bereits migriert. Der Codewegfall ergab dadurch keinen geringeren Migrationsaufwand. Aus Gründen der Wartbarkeit bleiben die Codezeilen aber weiterhin auskommentiert.

4.7. Migration verschiedener Datentypen

Dieses Kapitel beschreibt die Datentypen, die in der IML-Version verwendet wurden und in der *SKilL*-Version in dieser Weise nicht mehr zur Verfügung stehen. Dies betrifft die Ada-Arrays sowie die Datenstrukturen *Resizable_Array*, *Ordered_Sets* und *Hashed_Mappings*.

4.7.1. Migration Ada-Arrays und Array Aggregate

Serialisierte Arrays müssen in Ihre *SKilL*-Pendants, die *SkilL-Containers-Arrays*, konvertiert werden. Das *SkilL-Containers-Array* hält die Instanz des *SKilL-Containers-Vectors*. Das Vector-Paket stellt die Funktionalitäten eines Ada-Arrays zur Verfügung. Die rechte Spalte in Abbildung 4.3 listet alle bei der Migration benötigten Funktionen des Vector-Pakets auf. Ein indizierter Zugriff auf ein Feldelement erfolgt mit der Funktion *Element*. Die Größe des Feldes kann mit der Funktion *Length* abgefragt werden. Die restlichen gelisteten Methoden sind für die Repräsentation eines Ada-Arrays nicht relevant. Diese werden erst bei der in Kapitel 4.7.2 beschriebenen Konvertierung eines *Resizable_Arrays* benötigt.

In Ada können Arrays mithilfe von Aggregaten initialisiert werden. [ISO/IEC, 2012, Kapitel 4.3.3], [Barnes, 2014, Kapitel 8.2 und Kapitel 8.3] Der *SKilL*-Container unterstützt solche Aggregate nicht, weshalb semantisch äquivalente Funktionen des Container-Pakets verwendet werden müssen. Abbildung 4.2 zeigt einen Überblick über das Aggregat–Funktionsmapping. Zusätzlich zur Abbildung wird die verwendete Zuordnung im Folgenden detailliert erläutert.

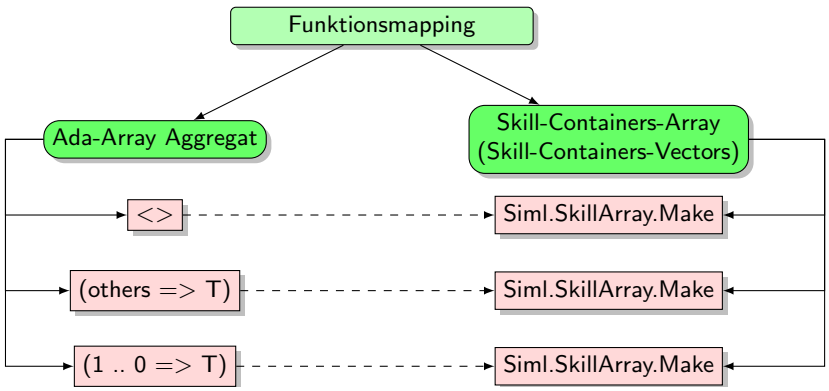

Abbildung 4.2: Funktionsmapping der Ada-Array Aggregate hin zu *SkilL-Containers-Arrays*

<> → Siml.SkillArray.Make Die *Box Notation* <> weist einer Variablen ihren Default-Wert zu [ISO/IEC, 2012, Kapitel 4.3.1], [Barnes, 2013, Kapitel 9.3.1]. Im Falle eines Arrays war in allen betrachteten Fällen der Anwendungsfall die Zuweisung eines Feldes innerhalb eines Records. Quellcode 8 zeigt einen solchen Aufruf anhand des Feldes *Aliasing*. (Zeile 5)

```
1 Context := new PTA_Context'
2              (Last_Param => Metrics (I).Last_Refparam,
3              ...
4              Actuals    => (others => (Is_Refparam => False)),
5              Aliasing   => <>);
```

Quellcode 8: Exemplarische Verwendung von Ada Aggregaten im IML-Code in der Datei *pta_objects.adb*

Die Konvertierung in die *SKilL*-Version muss nun im Einzelfall betrachtet werden. In den meisten Fällen kann die die Initialisierung mithilfe der *Make*-Funktion des *SKilL-Containers-Vectors* durchgeführt werden.

Besitzt das Feld jedoch einen abweichenden Default-Wert, muss dieser bei der Migration entsprechend nachgebildet werden. Quellcode 9 zeigt die tatsächliche Definition des Feldes *Aliasing*, das ein Feld der Länge *Last_Param* darstellt. Alle Elemente des Feldes sind mit dem Wert 0 initialisiert. Die *SKilL*-Nachbildung muss nun exakt diese Struktur nachbilden. Wichtig ist, dass dies bei jeder *Box Notation* <> gesondert betrachtet werden muss.

```
1 type PTA_Context (Last_Param : PTA_Offset) is limited
2   record
3      ...
4      Actuals     : Actual_Param_Targets (0 .. Last_Param);
5      Aliasing    : Alias_Class_Array (0 .. Last_Param) := (others =>
           0);
6 end record;
```

Quellcode 9: Auszug der Definition des Records PTA_Contex im IML-Code

Um das gezeigte Beispiel zu vervollständigen, zeigt Quellcode 10 die bereits genannte Initialisierung von *PTA_Context* aus Quellcode 8.

others => → Siml.SkillArray.Make Das Schlüsselwort OTHERS weist, falls benannte Aggregate vorausgingen, allen verbliebenen nicht befüllten Elementen den Wert T zu. Wird es ausschließlich verwendet, so werden alle Elemente mit dem Wert T initialisiert. Ähnlich wie bei der *Box Notation* <> muss auch hier die richtige Array-Größe beim *Make*-Aufruf verwendet werden. Die einzelnen Elemente können analog zu Quellcode 10 mithilfe einer FOR-Schleife initialisiert werden.

```
is
    TempAliasing : Siml.Skill_Array_v64.Ref;
begin
    TempAliasing := Siml.Skill_Array_v64.Make (Natural (Metrics (I).
        Last_Refparam));
    for K in 0 .. TempAliasing.This.Length - 1 loop
        TempAliasing.This.Replace_Element (K, 0);
    end loop;
    Context :=
        Skill_State.Get_Constructor_Graph.PTA_Context_Accesss.Make
            (F_Aliasing   => TempAliasing,
            ...
```

Quellcode 10: Initialisierung des Records PTA_Contex im *SKilL*-Code

1 .. 0 => → Siml.SkillArray.Make Mit dem *Null*-Aggregat (1 .. 0) wird in Ada ein *Null*-Array initialisiert. Der Typ T wird in diesem Fall vom Compiler nicht ausgewertet und hat somit an dieser Stelle keine Bedeutung. Das entsprechende *SKilL*-Pendant ist ein einfacher *Make*-Aufruf ohne Parameter des *SkilL-Containers-Array*.

Indizes: Bei der Indizierung ist es wichtig zu beachten, dass in Ada-Arrays nicht bei '0' starten, sondern mit jedem beliebigen Wert beginnen können. Die verwendeten *SKilL-Containers-Vectors* starten jedoch immer bei '0'. Bei der Migration kann nun gegebenenfalls eine Verschiebung erfolgen. Tabelle 3 zeigt die Konvertierungsregeln von den Attributen hin zu den entsprechenden *SKilL-Containers-Vectors* Ausdrücken.

Ada-Array-Attribute	*SKilL-Containers-Vectors* Ausdrücke
'First	0
'Last	This.Length - 1
'Length	This.Length
'Range	0 .. This.Length - 1

Tabelle 3: Konvertierung der Ada-Array-Attribute

Problematisch wird es, wenn im Code direkt mit numerischen Werten auf die Array-Elemente zugegriffen wird. Solche Zugriffe verursachen bei der Migration keine Compilerfehler und werden nicht entdeckt. Sollte nach der Migration auf Array-Elemente außerhalb des Index Bereichs zugegriffen werden, wird das Ada-Laufzeitsystem mit einer Ausnahme darauf hinweisen. Wird jedoch unglücklicherweise ein valider Indexwert verwendet, wird unwissentlich auf das falsche Array-Element zugegriffen. Kapitel 5.1 beschreibt einen solchen Fall.

4.7.2. Migration Resizable_Array

Im Bauhaus-Projekt existiert ein Paket *Resizable_Array*, eine Datenstruktur, die eine Art Liste nachbildet. Das *Resizable_Array* besitzt eine bestimmte Kapazität sowie eine

untere und obere Grenze. Die Variablen *First_Used* und *Last_Used* markieren die beiden Kapazitätsgrenzen. Innerhalb dieser existiert ein Teilbereich, der die gültigen Elemente beinhaltet. Der gültige Bereich startet bei *First_Used* und ist nach oben mit der Variable *Real_Used* ($<$ *Last_Used*) markiert. Das Paket besitzt über 50 Unterprogramme, mit welchen ein *Resizable_Array* bearbeitet werden kann. So kann zum Beispiel der gültige Bereich verändert werden oder Elemente hinzugefügt, gelöscht oder ausgetauscht werden. Auch ist ein sortiertes Einfügen möglich. Alternativ kann ein einmaliges Einfügen durchgeführt werden, um eine Menge bzw. ein Set zu simulieren. Ob ein *Resizable_Array* die Eigenschaft einer Menge besitzt, sortiert ist oder eine „einfache“ Liste widerspiegelt, kann pauschal nicht beantwortet werden. Dies ist Abhängig von der Verwendung der Unterprogramme.

In *SKilL* existiert kein Datentyp, der diese Eigenschaft besitzt. Stattdessen werden alle *Resizable_Arrays* in den in Kapitel 4.7.1 vorgestellten *SkilL-Containers-Array* umgewandelt. Der Array-Container verwaltet seine Elemente wiederum mit dem Paket *SKilL-Containers-Vectors*. Das Vector-Paket ist ein wesentlich übersichtlicher Datentyp als das *Resizable_Array* und orientiert sich mit seinem Funktionsspektrum an einer gängigen Liste. Der Vector startet immer bei dem Index ’0’ und endet bei dem letzten Index des letzten Elementes. Es gibt keine Kapazität mit einem gültigen und ungültigen Bereich, welche beide zur Laufzeit immer wieder angepasst werden müssen. Auch gibt es keine Möglichkeit Element sortiert einzufügen oder ein Set zu simulieren.

Abbildung 4.3 zeigt die vorgefundenen *Resizable_Array* Unterprogramme und die angewandte Zuordnung zu den Container-Methoden. Im Array-Container repräsentiert das Feld *This* den Vector, weshalb mit dem Zugriff .*This* die Vector-Operationen angewandt werden.

Include_Range (N) → This.Ensure_Allocation (N) Die Prozedur Include_Range stellt sicher, dass der angegebene Index N auch im gültigen Bereich des *Resizable_Arrays* liegt. Ist dieser kleiner wird der gültige Bereich vergrößert, gegebenenfalls wird auch die Kapazität vergrößert. Das eingesetzte Paket *SKilL-Containers-Vectors* besitzt keine ungültigen Indices und somit auch keinen Marker auf das letzte gültige Element. Jedoch kann mit der Prozedur *Ensure_Allocation* sichergestellt werden, dass der Vektor auch die Mindestgröße der benötigten Elemente zur Verfügung stellt.

Set_Range (N)→ This.Ensure_Allocation (N) Die Prozedur Set_Range modifiziert ebenfalls den gültigen Bereich des *Resizable_Arrays*. Im Gegensatz zu Include_Range, welches den Bereich nur bei Bedarf vergrößert, wird er hier explizit auf Größe N gesetzt. Auf diese Weise ist auch eine Reduktion des gültigen Bereichs möglich. Die Kapazität des *Resizable_Array* wird beim Verkleinern nicht angepasst. Mit dem bereits beschriebenen Ensure_Allocation kann auch hier sichergestellt werden, dass ausreichend viele Indices allokiert sind. Schwieriger wird jedoch der Fall des Verkleinerns. Die ursprüngliche Idee hinter den gültigen und ungültigen Bereichen eines *Resizable_Array* konnten nicht nachvollzogen werden. Es ist nicht klar, ob ein ungültiger Bereich zu einem späteren Zeitpunkt wieder gültig sein kann. Diese Information ist jedoch mit Hinblick auf ein

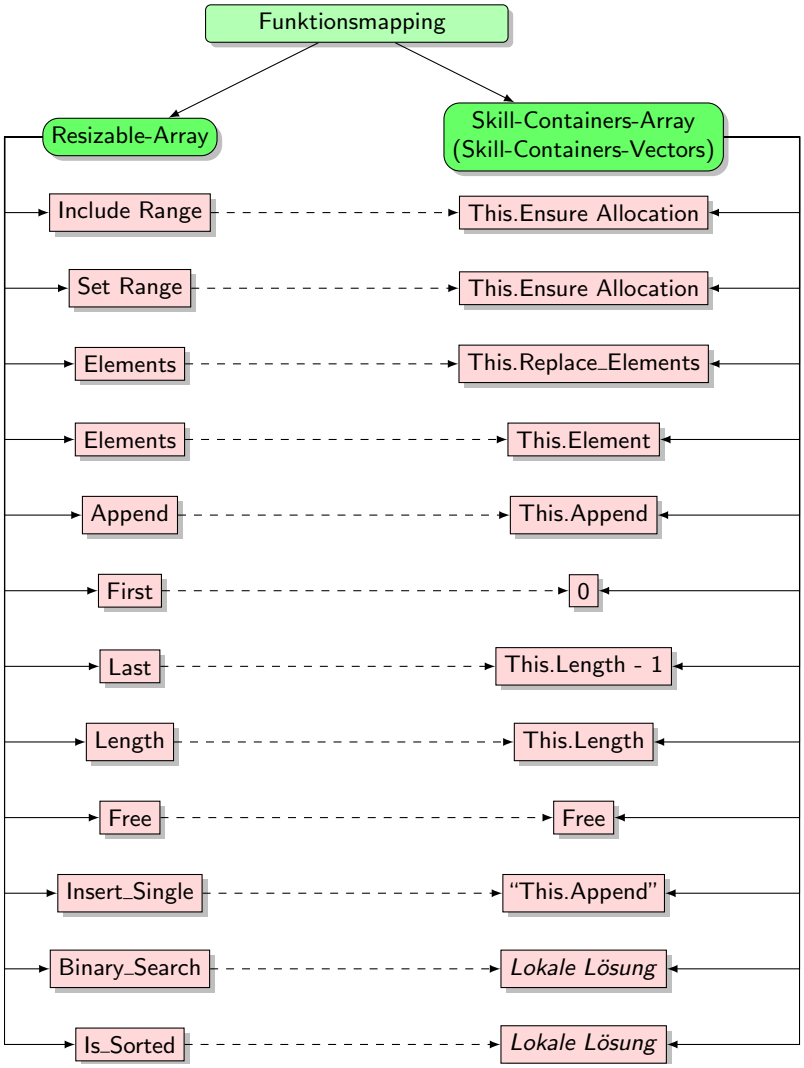

Abbildung 4.3: Funktionsmapping *Resizable_Array* zu *SkilL-Containers-Array*

mögliches Verkleinern essentiell. Ist das Reduzieren des gültigen Bereichs äquivalent zu einem Löschen der letzten Elemente, könnte in der *SKilL*-Version mit der Funktion *pop* gearbeitet werden. Es bleibt fraglich, warum dieser Mechanismus überhaupt existiert und warum nicht gleich die Kapazität verringert wird. Hierfür wird nämlich ebenfalls eine *Resizable_Array* Prozedur *Shrink* angeboten. Im anderen Fall müsste der *SKilL*-Datentyp ebenfalls einen Zeiger erhalten, um somit ähnliches Verhalten nachzubilden.

Das primäre Ziel dieser Arbeit ist es, einen compilierbaren und ausführbaren Softwarestand zu erreichen. Aus diesem Grund wurde im Rahmen der Migration Set_Range im ersten Schritt mit Ensure_Allocation ersetzt. Auf diese Weise sind zur Laufzeit ausreichend viele Elemente allokiert und die Software ist hinsichtlich Set_Range ausführbar. Die inhaltlichen Auswirkungen können im Anschluss bei den Laufzeittests genauer untersucht werden. Gegebenenfalls muss dieser Migrationsschritt dann angepasst werden.

Elements (N) → This.Element (N) Ein indizierter Zugriff auf die Elemente des *Resizable_Arrays* erfolgt über das Feld *Elements*. Hierbei handelt es sich um ein Ada-Array vom Typ der zu speichernden Werte. Auf diese Weise können Elemente direkt ersetzt oder gelesen werden. Ein Lesezugriff beim *SKilL-Containers-Vectors* erfolgt hingegen über die Funktion *Element (N)*. Wichtig ist, dass N vom Typ *Integer* ist und gegebenenfalls ein Cast notwendig ist. Ebenfalls zu beachten ist, dass der Elementtyp des Vectors und der erwartete Rückgabetyp der Funktion identisch sind. Ansonsten ist wieder ein Cast oder eine Typ-Umdefinition erforderlich. Dies wird an dieser Stelle explizit erwähnt, da der Compiler diesen Fehler nicht aussagekräftig auflöst, sondern zurückmeldet, dass der Selector *Element* nicht existiert.

Elements (N) → This.Replace_Element (N, E) Analog zum Lesezugriff wird der Schreibzugriff mit der Prozedur Replace_Element durchgeführt. Der Parameter N ist der Index, E das Element. Auch hier sind die exakten Typen erforderlich, um einen Compilerfehler zu vermeiden.

Append (E, Exp_Grow) → This.Append (E) Beide Datenstrukturen stellen eine Append Funktionalität zur Verfügung. Die Prozedur des *Resizable_Arrays* erwartet zusätzlich einen zweiten Booleschen Parameter *Exp_Grow*. Dieser entscheidet im Falle einer Kapazitätserhöhung, ob sich die Kapazität des Arrays um einen Faktor von mindestens 3/2 erhöht. Dadurch sollen Performanzgewinne bei folgenden Append Aufrufen erreicht werden. Wie bereits erläutert unterscheidet der *SKilL*-Datentyp nicht zwischen einem gültigen und ungültigen Bereich. Bei der Migration kann der zweite Booleschen Parameter somit ignoriert werden.

First → 0 First gibt den kleinsten Index des gültigen Bereiches zurück. In aller Regel starteten die migrierten *Resizable_Arrays* mit dem Index '1'. Wie in Kapitel 4.7.1 bereits erläutert, beginnen alle *SKilL*-Container Typen immer mit dem Index '0'. Aus diesem Grund wird auch keine äquivalente First Implementierung zur Verfügung gestellt, sondern direkt '0' verwendet.

Length → This.Length Beide Datenstrukturen enthalten eine Funktion Length, die die Anzahl der enthaltenen Elemente zurück gibt. Im Falle des *Resizable_Array* bezieht sich die Anzahl auf die Elemente im gültigen Bereich.

Last → This.Length - 1 Analog zu First gibt Last den größten Index des gültigen Bereiches zurück. Da der *SKilL-Containers-Vectors* bei '0' beginnt, definiert die um eins verringerte Länge den größten verfügbaren Index des Vectors.

Free → Free Mittels der Funktion Free kann ein *Resizable_Array* deallokiert werden. Hierzu wird die Ada Prozedur *Ada.Unchecked_Deallocation* verwendet. Im Rahmen

dieser Arbeit wurden alle *SKilL*-Container um selbige Free-Funktionalität erweitert. Im Gegensatz zu den bisher vorgestellten Funktionszuordnungen wird jedoch nicht mit *This* auf den Vector zugegriffen, sondern das Free des *SkilL-Containers-Array* verwendet.

Insert_Single → "This.Append" Zu Beginn dieses Kapitels wurde beschrieben, dass ein *Resizable_Array* auch die Eigenschaften eines Sets besitzen kann. Hierfür wird die Prozedur Insert_Single verwendet. Mit dieser wird ein Element in ein vorsortiertes *Resizable_Array* eingefügt, sofern das Element noch nicht vorhanden ist. Da ein *SkilL-Containers-Array* kein Set ist, wird eine solche Funktionalität nicht unterstützt. Für die Migration ergeben sich mehrere Möglichkeiten:

Um die verbleibenden bereits diskutierten *Resizable_Array* Funktionen und ihre Zuordnung zu *SKilL*-Pendants im *SkilL-Containers-Array* zu erhalten, kann man die Set-Eigenschaften im ersten Ansatz ignorieren und stattdessen die vorhandene Append Funktion verwenden. Im besten Fall wird das Set nur aus Performanzgründen benötigt, um zum Beispiel eine schnellere Suche von Elementen zu ermöglichen.

Alternativ kann man auch betroffene Variablen in ein *SkilL-Containers-Set* konvertieren. Dieser Datentyp erhielte die Set-Eigenschaft, würde aber eventuell weitere benötigte *Resizable_Array* Funktionalitäten nicht unterstützen. Prinzipiell stellt sich hier die Frage, wieso bei der Notwendigkeit eines Sets anstatt eines *Resizable_Arrays* nicht der ebenfalls im Projekt weit verbreitete Datentyp *Ordered_Sets* verwendet wurde. Die Migration dieses IML-Sets hin zu dem erwähnten *SKilL*-Set ist im folgenden Kapitel 4.7.3 beschrieben.

Eine Nachimplementierung des Insert_Single im *SkilL-Containers-Array* wurde bewusst unterlassen. Ein Set und ein Array sind zwei getrennte Datentypen und sollen auch als solche behandelt werden. Als eine etwas abgewandelte Form dieses Vorschlages könnte man auch ein neues Wrapper-Paket implementieren, welches ein *SkilL-Containers-Array* verwaltet und zusätzlich die Set-Eigenschaft herstellt.

Im Kontext der Migration wurde der erste Ansatz verfolgt, wohl wissend, dass es eventuell zu inhaltlichen Abweichungen kommen kann. Ähnlich wie in Kapitel 4.7.2 bei der Konvertierung der Funktion Set_Range, wird auch hier eine vereinfachende Annahmen zugunsten einer einfacheren Migration getroffen. Bei einer Auswirkung auf das Programmverhalten kann diese Entscheidung gegebenenfalls revidiert werden. Da einzig die Variable *Deref_Pair_Arrays* von dieser Vereinfachung betroffen ist, bleibt das Risiko und der eventuelle Anpassungsaufwand überschaubar.

Is_Sorted → Lokale Lösung Die Funktion Is_Sorted überprüft, ob ein *Resizable_Array* sortiert ist. Der Aufruf von Is_Sorted erfolgt in der IML-Implementierung nur einmalig innerhalb einer *Assert*-Überprüfung. Aus diesem Grund und da ein *SkilL-Containers-Array* in der Regel nicht sortiert ist, wurde die *SKilL*-Datenstruktur nicht mit einer äquivalenten Funktion erweitert. Stattdessen wurde eine lokale Funktion Is_Vector_Sorted implementiert, welche die benötigte Funktionalität zur Verfügung

stellt. Die Implementierung findet man im Paket *PTA_DF_Pattern*. Während den Laufzeittests wurde die Assert-Überprüfung mit Is_Vector_Sorted nicht ausgelöst.

Binary_Search → Lokale Lösung Das *Resizable_Array* enthält eine generische Prozedur General_Binary_Search, die als formalen Parameter eine Vergleichsfunktion Compare erwartet. Mit Hilfe der Vergleichsfunktion wird eine binäre Suche auf dem Array durchgeführt. Die binäre Suche erfolgt ausschließlich auf dem Datentyp *Locator_Value*, der auch als Beispiel für die Migration eines *variant record* in Kapitel 4.9 verwendet wird. Je nach zu suchendem Element werden unterschiedliche Comparefunktionen verwendet, um die Suche zu optimieren. Der Rückgabewert der Comparefunktion, ein Enumerator, gibt der binären Suche den Hinweis, in welche Richtung weiter gesucht werden muss.

Da das *SkilL-Containers-Array* nicht sortiert ist, kann auch keine binäre Suche angewendet werden. Aus gleichem Grund ist auch eine Nachimplementierung in der Datenstruktur nicht erstrebenswert. Stattdessen wurde ähnlich wie bei Is_Sorted eine lokale Lösung implementiert. Die verschiedenen Comparefunktionen geben nur noch einen Booleschen Wert zurück. *True* falls das Element gefunden wurde, ansonsten *False*. Die Suche erfolgt anschließend mit einer einfachen FOR-Schleife, die über das *SkilL-Containers-Array* iteriert und bei jedem Element die Comparefunktion aufruft. Quellcode 11 veranschaulicht eine solche lineare Suche.

```
1 for Index in 0 .. Pattern.Get_Variables.This.Length - 1
2 loop
3     if Compare (Element => Pattern.Get_Variables.This.Element (Index)
          .Dynamic_Locator_Value) then
4         return Locators.Locator (Index); -- found
5      end if;
6 end loop;
7 return Locators.No_Locator; -- not found.
```

Quellcode 11: Vereinfachte Suche als Ersatz für die binäre Suche der *Resizable_Arrays*

Die Komplexität des Suchalgorithmus vergrößert sich durch diese Vereinfachung von $O(log(n))$ hin zu $O(n)$. Da die Größenordnung von n nicht bekannt ist, kann keine Aussage über die Laufzeitauswirkung getroffen werden. Bei den Testdurchläufen, die in Kapitel 6 beschrieben sind, wurden jedoch selbst bei großen Dateien keine negativen Auswirkungen beobachtet.

4.7.3. Migration Ordered_Sets

Weiterhin existiert im Bauhaus-Projekt die Datenstruktur *Ordered_Sets*, welche ebenfalls migriert werden muss. Das Paket *Ordered_Sets* verwaltet intern einen AVL-Baum. Das *SKilL*-Pendant ist das *SkilL-Containers-Set*. Analog zum *SkilL-Containers-Array* besitzt der Container ein Feld *This*, mit dem auf ein Ada *Hashed_Sets* zugegriffen werden kann. Das *Hashed_Sets* stellt den benötigten Funktionsumfang des *Ordered_Sets*

zur Verfügung. Einzig die Sortiereigenschaft des AVL-Baumes kann so nicht dargestellt werden. Negative Einflüsse auf die Performanz konnten bei den Laufzeittest bisher nicht beobachtet werden. Ähnlich wie bei der Umsetzung der Funktion Insert_Single des *Resizable_Array* in Kapitel 4.7.2 konnten inhaltliche Abweichungen beim Abschluss dieser Arbeit noch nicht festgestellt werden. Abbildung 4.4 zeigt die erarbeiteten Konvertierungsregeln in einer Übersicht. Die Überlegungen sind in den folgenden Paragraphen begründet.

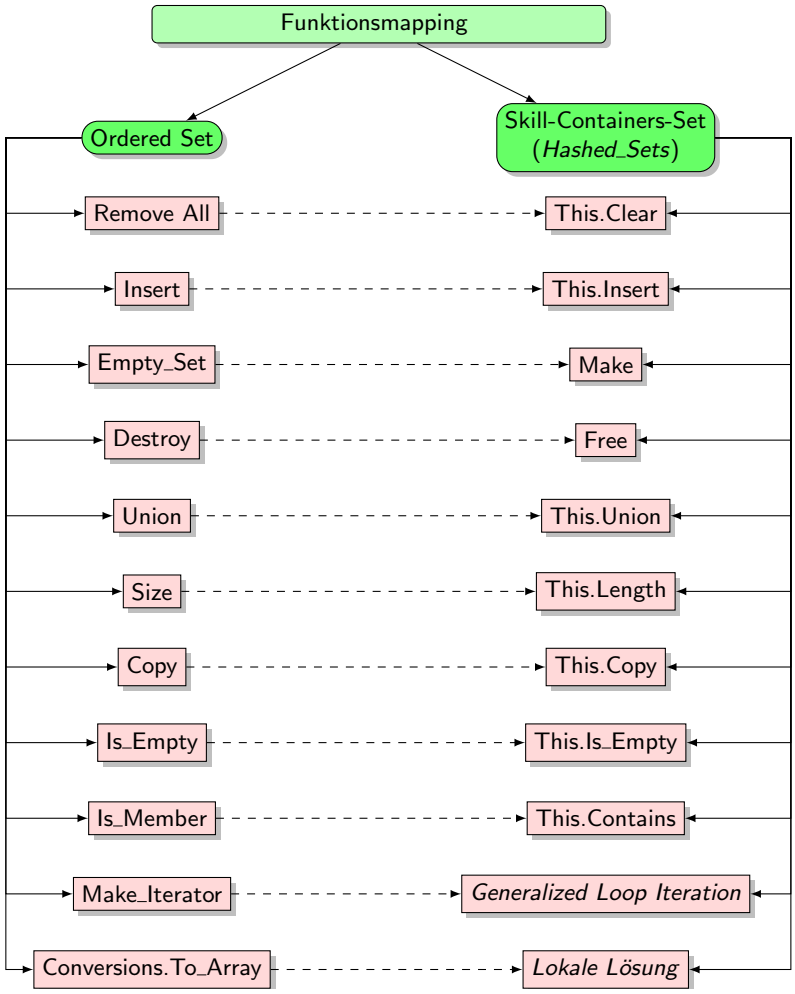

Abbildung 4.4: Funktionsmapping von *Ordered_Sets* zu *SkilL-Containers-Array*

Remove_All → This.Clear Beide Sets besitzen jeweils eine Prozedur zum Löschen aller Elemente des Sets: Remove_All im *Ordered_Sets* und This.Clear im Ada-*Hashed_Sets*.

Insert → This.Insert Beide Datenstrukturen stellen jeweils zwei Insert Prozeduren zur Verfügung. Im *Ordered_Sets* wird das Element in den verwalteten AVL-Baum ein-

gefügt, das *SkilL-Containers-Set* verwendet hingegen das *Hashed_Sets*. Die jeweils erste Insert Prozedur in jedem Set erwartet einen Booleschen Out Parameter, der indiziert ob das Element bereits im Set vorhanden ist. Die jeweils zweite Prozedur erwartet neben dem eigentlichen Set nur das einzufügende Element als Parameter. Im Gegensatz zum *Ordered_Sets* kann bei dieser Insert Prozedur des *Hashed_Sets* eine Ausnahme ausgelöst werden, sollte das einzufügende Element bereits vorhanden sein. Diese Ausnahme wurde aber bei den durchgeführten Laufzeittests nicht ausgelöst.

Empty_Set → Make Die Funktion *Make* des *SkilL-Containers-Set* initialisiert das eigene *Hashed_Sets* mit einem leeren Set und kann somit als Gegenstück zu Empty_Set von *Ordered_Sets* verwendet werden.

Destroy → Free *Destroy* iteriert rekursiv über den Baum und führt eine *Unchecked_Deallocation* für jeden Knoten aus. Analog besitzt das *SkilL-Containers-Set* eine Free Version, die das Set deallokiert.

Union → This.Union Die *Union* Prozedur von *Ordered_Sets* vereint zwei Sets. Das erste Set wird dabei modifiziert, das zweite bleibt erhalten. Die äquivalente Funktion *Union* der *Hashed_Sets* stellt dieselbe Funktionalität bereit.

Size → This.Length Die Anzahl der Elemente kann über die *Hashed_Sets* Funktion *Length* abgefragt werden. Falls noch nicht vorhanden, muss der entsprechende *use clause* für *Ada.Containers.Count_Type* im Deklarationsblock hinzugefügt werden. Ansonsten könnten eventuell benötigte Operatoren nicht sichtbar sein.

Copy → This.Copy Das *Hashed_Sets* des Skill-Containers stellt ebenfalls eine äquivalente *Copy* Funktion zur Verfügung. Problematisch ist oftmals aber, dass keine Kopie des *Hashed_Sets* gefordert ist, sondern eine Kopie des kompletten *SkilL-Containers-Set* inklusive des *Hashed_Sets*. In diesem Fall muss die *Copy* Funktion mit einem *Make*-Aufruf verknüpft werden. Das kopierte *Hashed_Sets* kann nun mittels eines *Union* Aufrufes in den neuen Container integriert werden.

Make_Iterator → Generalized Loop Iteration Die Funktion *Make_Iterator* gibt einen Iterator über das komplette Set zurück. *SkilL-Containers-Set* stellt ebenfalls eine Funktion Iterator zur Verfügung, die einen Cursor auf das *Hashed_Sets* zurück gibt. Mit Hilfe des Cursors kann man nun ebenfalls über das Set iterieren. Oftmals kann aber auch komplett auf die Verwendung eines separaten Iterators verzichtet werden. Ada 2012 führte hierzu die *Generalized Loop Iteration* [ISO/IEC, 2012, Kapitel 5.5.2], [Barnes, 2014, Kapitel 21.6] ein. Ein entsprechendes Beispiel ist in Kapitel 4.12.2 gelistet.

Conversions.To_Array → Lokale Lösung Das *Ordered_Sets* bietet die Möglichkeit das Set in ein Ada-Array zu konvertieren. Das *SkilL-Containers-Set* stellt eine solche Funktionalität nicht zur Verfügung. Da diese nur in einer einzigen Datei, nämlich in *pta_df_pattern.adb*, benötigt wird, wurde sich für eine lokale Lösung entschieden. Mithilfe der *Generalized Loop Iteration* wird über das *Hashed_Sets* iteriert. Die Elemente werden in ein neu allokiertes Ada-Array gespeichert.

4.7.4. Migration Hashed_Mappings

Eine weitere zu migrierende Datenstruktur ist das Paket *Hashed_Mappings*. Das verwendete *SKilL*-Äquivalent ist das *SkilL-Containers-Map*, das wiederum die Ada *Hashed_Maps* verwaltet. Die beiden einzigen verwendeten Unterprogramme von *Hashed_Mappings* sind Bind und Fetch. Diese lassen sich sehr gut mit dem *SKilL*-Container darstellen. Abbildung 4.5 zeigt die verwendete Funktionszuordnung.

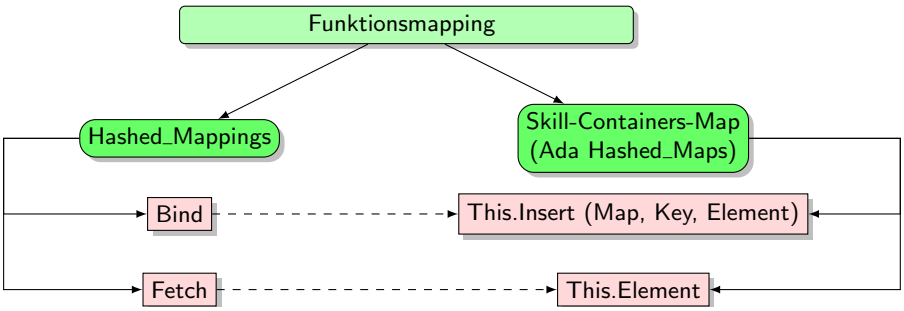

Abbildung 4.5: Funktionsmapping von *Hashed_Mappings* zu *SkilL-Containers-Map*

Bind → This.Insert (Map, Key, Element) Die Prozedur *Bind* fügt ein übergebenes Schlüssel-Werte-Paar in die Map ein. Eine *Already Bound* Ausnahme wird geworfen, falls ein solches Paar bereits in der Map vorhanden ist.

Die von *SKilL* genutzte *Hashed_Maps* bietet drei verschiedene Insert Methoden an. Je nach Parameteranzahl wird mit dem Out Parameter *Inserted* indiziert, ob ein Paar erfolgreich eingefügt wurde. Zusätzlich kann mit einem Out Parameter *Position* auch noch die Position des gerade eingefügten Paares (oder des bereits vorhandenen Paares) ermittelt werden. Die *Insert* Prozedur mit nur drei Parametern, der Map, dem Schlüssel und dem Wert, entspricht der beschriebenen *Bind* Prozedur. Auch hier wird im Falle einer Doppelbelegung eine Ausnahme geworfen.

Fetch → This.Element Das Paket *Hashed_Mappings* stellt sowohl eine Funktion als auch eine Prozedur *Fetch* zur Verfügung. In beiden Methoden wird der korrespondierende Wert zum übergebenen Schlüssel zurückgegeben oder im Falle der Prozedur als Out Parameter gesetzt. Die Funktion wirft eine *Not Bound* Ausnahme, falls das gesuchte Schlüssel-Werte-Paar nicht existiert. Die Prozedur setzt in diesem Fall hingegen nur den Booleschen Out Parameter *Found* auf *False*.

Das Pendant der Ada *Hashed_Maps* wäre hier die Funktion *Element*, die einen Schlüssel als Parameter erwartet und das passende Element zurück gibt. Ist der Schlüssel nicht in der Map, wird auch hier eine Ausnahme ausgelöst.

4.8. Auflösen eines Aufzählungstyps

Im *SKilL*-Zustand werden Enum-Variablen als *Integer_64* verwendet. Die Definition des Aufzählungstyps bleibt aus Gründen der Lesbarkeit im betroffenen Paket erhalten, der Enum-Typ kann somit weiterhin verwendet werden. Dies bedeutet jedoch, dass beim Ein– und Auslesen der serialisierten Variablen die entsprechenden Ada Attribute 'Pos und 'Val verwendet werden müssen. Bei dem 'Val Attribut ist zusätzlich noch ein Cast zu *Integer* erforderlich. Selbiges Verfahren wurde auch in der Vorgängerarbeit [Przytarski, 2016, Kapitel 4.4.4] angewandt. Der Pseudocode in Quellcode 12 zeigt ein Beispiel, wie mit den Enum-Werten umgegangen werden kann. Angenommen in einer *.ads*-Datei wird ein Enumerator *Farbe_Enum* mit den Werten *Rot* und *Blau* definiert. Im Rahmen der Migration, existiert nun in der *SKilL*-Spezifikation und somit auch im Ada-Binding eine Instanz *Farbe* vom Typ *V64*. Die Pseudovariable *Bar* besitzt ein Feld vom Typ *Farbe*. Alle Namen sind beliebig gewählt.

```
1 if Farbe_Enum'Val (Integer (Bar.Get_Farbe)) = Rot then
2     Bar.Set_Farbe (Farbe_Enum'Pos (Blau));
```

Quellcode 12: Pseudocode zum Ein- und Auslesen von ehemaligen Enum-Typen

4.9. Auflösen eines Variant Records

Im Bauhaus-Code werden unter anderem *variant records* [ISO/IEC, 2012, Kapitel 3.8.1] verwendet. *Variant records* lassen sich mit *Unions* aus der C-Welt vergleichen. Je nach Unterscheidungsvariable (der Diskriminante) stellt der Record verschiedene Varianten, sprich Felder mit verschiedenen Typen, zur Verfügung. *SKilL* unterstützt keine *variant records*, weshalb ein Benutzertyp verwendet wird, der alle Felder in einer Hierarchie vereint. [Przytarski, 2016, 10.2]

Für die Migration der *variant records* in den zu migrierenden Dateien sind einige Umwandlungsschritte erforderlich. Die Zugriffe auf die neuen Typen werden dabei mit einer neuen Hilfsfunktion realisiert. Alle notwendigen Änderungen werden im Folgenden exemplarisch an der Umwandlung des Typs *Locator_Value* erläutert. Quellcode 13 zeigt die Definition des *variant records Locator_Value* im Paket *PTA_DF_Pattern* in der IML-Version.

Dieser Typ wird durch das generierte Pendant des Ada-Bindings ersetzt und somit im Paket gelöscht. Um den Namen des Typs nicht zu verlieren, wurde wie zuvor wieder ein neuer Untertyp eingeführt. Siehe hierzu Quellcode 14.

```
type Locator_Management is
 (Local_Refparam,   --  can be mapped back to caller, needs offset
  Local,            --  only in one single pattern (local variable)
  Global,           --  mapping using pta map
  Return_Value);    --  no mapping required for return values

type Locator_Value (Kind : Locator_Management := Global) is
  record
     case Kind is
        when Local | Global =>
           Object : PTA_Objects.PTA_Object := PTA_Objects.
              No_PTA_Object;
        when Local_Refparam =>
           Number : Integer;
        when Return_Value =>
           null;
     end case;
end record;
```

Quellcode 13: IMLBindeStrich Definition von *Locator_Value*

```
subtype Locator_Value is Siml.Locator_Value_Dyn;
subtype Locator_Value_Class is Siml.Locator_Value_T;
```

Quellcode 14: *SKilL* Subtype Definition von *Locator_Value*

Der neue *SKilL*-Typ *Locator_Value* ist dabei nicht mehr als *variant record* implementiert. Die ehemaligen Record Feldertypen *Local*, *Global*, *Local_Refparam* und *Return_Value* sind jetzt eigene Instanzen in einem hierarchischen Aufbau. Abbildung 4.6 illustriert die neue Datenstruktur *Locator_Value*.

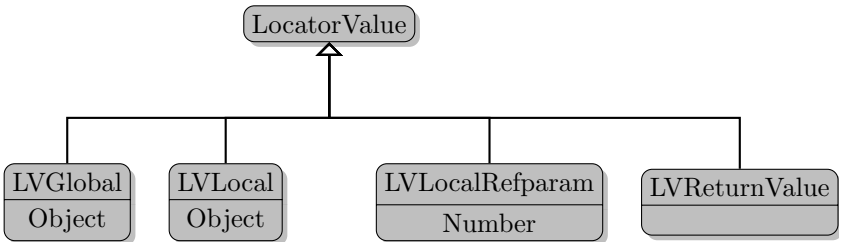

Abbildung 4.6: Neuer hierarchischer Aufbau von *Locator_Value*

Der Codeausschnitt Quellcode 15 zeigt einen bis dahin gültigen Zugriff auf die Variable *Left* vom Typ *Locator_Value*.

Die Funktion $<$ definiert in dem Beispiel eine Ordnung für die Enum-Typen von *Locator_Value*.

```
Return_Value < Local_Refparam < {LVLocal, LVGlobal}
```

```
function "<"
 (Left  : in Locator_Value;
  Right : in Locator_Value)
 return Boolean
is
...
begin
  case Left.Kind is
     when Return_Value =>
        ...
     when Local_Refparam =>
        case Right.Kind is
           when Return_Value =>
              return False;
           when Local_Refparam =>
              return Left.Number < Right.Number;
           when Local | Global =>
                    return True;
        end case;
     ...
  end case;
end "<";
```

Quellcode 15: Beispielhafter IML-Zugriff auf *Locator_Value*

Bei Typgleichheit wird der numerische Wert verglichen. Betrachtet man die Syntax, ergeben sich bei der *SKilL*-Migration folgende Probleme:

Zum einen kann nicht mehr mit der Punktnotation auf die Felder zugegriffen werden. Stattdessen müssen wie in Kapitel 4.5 beschrieben, die entsprechenden Getter- und Setter-Methoden verwendet werden. Weiterhin ist eine einfache Abfrage nach dem Enum-Typ, wie zum Beispiel in der Switch-Case Anweisung realisiert, nicht mehr möglich. Da es sich um eigenständige Instanzen handelt, ist hierzu eine Typabfrage mit anschließendem Cast notwendig. Um den Anpassungsaufwand zu minimieren, wurde eine Hilfsfunktion implementiert, die die Enumabfrage simuliert. Diese neue Funktion *Kind* in Quellcode 16 gibt als Return Value den selben Enum-Typ wie bei der Verwendung des ursprünglichen *variant record* zurück und kann an allen Aufrufstellen von *Locator_Value* verwendet werden. Anstelle der Punktnotation in Quellcode 15 erfolgt stattdessen der Funktionsaufruf mit der neuen Funktion *Kind*.

Bevor die richtige Getter-Methode (*Get_Number* oder *Get_Object*) aufgerufen werden kann, muss der *Locator_Value* auf den richtigen Typ gecastet werden. Aufgrund der Switch-Case Abfrage ist dieser zu diesem Zeitpunkt bereits bekannt. Analog zum Codeauschnitt in Quellcode 17 sind in der gesamten Applikation die Zugriffe auf den Datentyp *Locator_Value* angepasst worden.

Im gezeigten Beispiel lässt sich ein weiterer Effekt beobachten. In Abbildung 4.6 erkennt man, dass beide Typen *LVGlobal* und *LVLocal* ein Feld Namens *Object* enthalten. Im ursprünglichen IML-Code wird auf diese Felder direkt mit der Punktnotation zugegrif-

```
1  function Kind (This : Locator_Value)
2                 return Locator_Management
3  is
4    Type_name : constant Skill.Types.String_Access :=
5      This.Skill_Name;
6    use type Skill.Types.String_Access;
7  begin
8    if Type_name = Siml.Internal_Skill_Names.Lvlocal_Skill_Name then
9       return Local;
10   elsif Type_name = Siml.Internal_Skill_Names.Lvglobal_Skill_Name
         then
11      return Global;
12     ...
13    end if;
14 end Kind;
```

Quellcode 16: Neue Hilfsfunktion *Kind*, welche das ursprüngliche Enum Verhalten von *Locator_Value* simuliert

```
1 ...
2 case Kind (Left) is
3 ...
4      when Local_Refparam =>
5        return Left.As_LV_Local_Refparam.Get_Number < Right.
             As_LV_Local_Refparam.Get_Number;
6 ...
```

Quellcode 17: Migriertes Beispiel aus Quellcode 15. Die Variable *Locator_Value* wird nun mit Hilfe der neuen Hilfsfunktion *Kind* verwendet. Bevor auf das Feld zugegriffen werden kann, muss der richtige Cast verwendet werden.

fen. Oftmals geschieht dies nach einer logischen ODER-Abfrage ähnlich wie in Zeile 17 in Quellcode 15 oder wenn es vom Programmkontext ersichtlich ist, dass es sich um ein *Local* oder *Global* handelt. Um das gleiche Verhalten in der *SKilL*-Version nachzubilden, wurde die Spezifikation erweitert. Eine neue Zwischeninstanz *LVGlobalLocal* enthält nun das Feld *Object*. Die beiden Typen *LVLocal* und *LVGlobal* erben von dem neuen Typ. Abbildung 4.7 illustriert die neue Struktur. Die hierzu notwendigen Anpassungen der *SKilL*-Spezifikation sind in Kapitel 4.10.2 dokumentiert.

Neben der hier ausführlich beschriebenen Umwandlung von *Locator_Value* wurde auf ähnliche Weise auch der *variant record Refparam_Target* im Paket *PTA_Object* angepasst. Anstatt eines Enums verwendet diese Variable jedoch einen Booleschen Wert als Diskriminante.

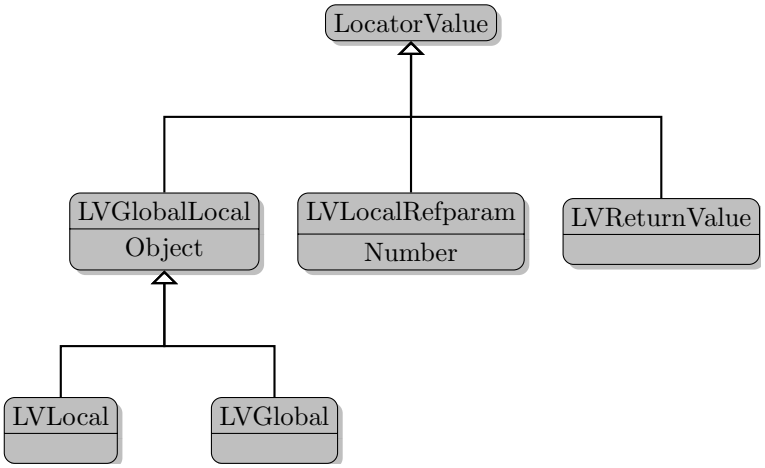

Abbildung 4.7: Verbesserter hierarchischer Aufbau von *Locator_Value*

4.10. Anpassung der Spezifikation

Im Rahmen dieser Arbeit wurde vereinzelt auch die *SKilL*-Spezifikation angepasst. Alle Spezifikationsanpassungen gegenüber dem Ausgangsstand sind in diesem Kapitel festgehalten. Die Änderungen erfolgten alle ausschließlich am Spezifikationsgenerator specgen, der die *SKilL*-Spezifikation generiert.

4.10.1. Fehlende Hierarchie: Storable

Wie in Kapitel 2 beschrieben, definiert die Spezifikation einen hierarchischen Objektaufbau. Die generierten Klassen erben alle direkt oder indirekt von der abstrakten Klasse IML_Root, die wiederum von *Storable* erbt. Die Klasse *Storable* definiert im Wesentlichen Lese– und Schreibmethoden, die von allen Unterklassen implementiert werden müssen. [Przytarski, 2016, S.11] Den beiden Objekten *Static_Predicates* und *DF_Pattern* fehlten im vorliegenden Ada-Binding die hierarchische Zuordnung, was bei der Übersetzung des pta_tools zu Fehlern führte. Zur Lösung dieses Problems wurde der Spezifikationsgenerator specgen in der Datei *imlgen4/src/main/scala/specgen/-SkillBuiltinsMaker.scala* angepasst. Die benötigte Vererbung wird in *SKilL* mit dem Schlüsselwort *extends* realisiert. Eine ausführliche Beschreibung der *SKilL*-Syntax kann [Felden, 2017, S.11] entnommen werden.

4.10.2. Anpassung Locator_Value

In Kapitel 4.9 wurden notwendige Anpassungen am hierarchischen Aufbau des Typs *Locator_Value* diskutiert. Quellcode 18 zeigt die Änderungen die vorgenommen wurden, um dieses Ziel zu erreichen.

```
+ LV_GlobalLocal : Locator_Value {
+ PTA_Object Object;
+ }
 !pragma inPackage(Locator_Values)
- LV_Global : Locator_Value {
-  PTA_Object Object;
+ LV_Global : LV_GlobalLocal {
 }
 !pragma inPackage(Locator_Values)
- LV_Local : Locator_Value {
- PTA_Object Object;
+ LV_Local : LV_GlobalLocal {
```

Quellcode 18: Der Spezifikationsgenerator specgen wurde in der Datei *src/main/-scala/specgen/SkillBuiltinsMaker.scala* angepasst, um der Hierachie aus Abbildung 4.7 zu entsprechen.

4.10.3. Fehlender Typ Call_Site_Callees

In der *SKilL*-Spezifikation war das Feld *Call_Mapping* des Typs *PTA_DF_Pattern* fälschlicherweise ein Array vom Typ *Call_Site_Callee*. In der IML-Implementierung war das Feld jedoch vom Typ *Call_Site_Callees* (mit s). *Call_Site_Callees* ist wiederum ein *Resizable_Array* vom Typ *Call_Site_Callee*. Der *SKilL*-Spezifikationsgenerator specgen wurde entsprechend angepasst. Das Diagramm in Abbildung 4.8 zeigt die Hierarchie mit dem neuen *SkilL-Containers-Array Call_Site_Callees*. Um die Verschachtelung der beiden *SkilL-Containers-Arrays* darzustellen, wurde das Feld *data* definiert.

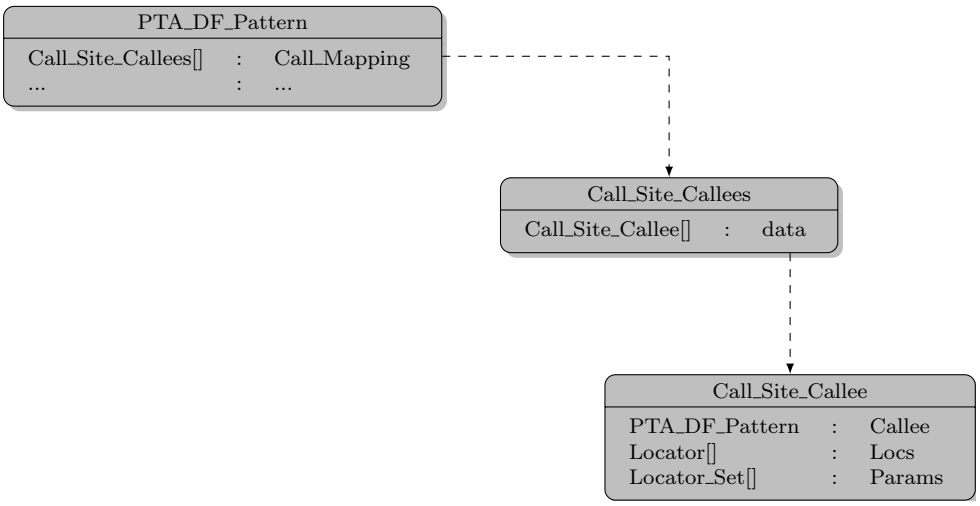

Abbildung 4.8: Die *SKilL*-Spezifikation wurde mit dem Typ *Call_Site_Callees* erweitert, um der IML-Implementierung zu entsprechen.

4.11. Anpassungen des Codegenerators

Dieses Kapitel enthält Anpassungen am Codegenerator codegen, die im Rahmen der Migration notwendig waren. Bei den gelösten Problemen handelte es sich nicht um wiederkehrende Fehlermuster. Vielmehr musste jeder Fehlerfall individuell betrachtet und gelöst werden. Aufgrund dessen ist es auch nicht zielführend, jede vorgenommene Codegeneratoränderung in der vorliegenden Arbeit zu dokumentieren. Vielmehr sollen im Folgenden repräsentative Beispiele für diese Fehlertypen gegeben werden.

4.11.1. Fehlerhaft generierte Set-Prozeduren

Quellcode 19 zeigt einen Codeausschnitt einer generierten Setter-Prozedur in *values.adb* aus libiml. Beim Ausführen der Funktion wird zwangsläufig die Ausnahme in Zeile 10 ausgelöst. Vielmehr ist hier ein verschachteltes IF-ELSIF-ELSE Konstrukt gewollt, bei welchem das finale ELSE die Ausnahme beinhaltet. Dieser fehlerhafte Code wurde nicht nur bei der vorliegenden Setter-Methode, sondern bei allen Feldzugriffen in *values.adb* generiert. Der Codegenerator wurde entsprechend angepasst, sodass ein IF-ELSIF-ELSE generiert wird.

```
1  procedure Set_PTA_Object (...) is
2  begin
3    if Node.all in Siml.Address_Of_T'Class then
4       Node.As_Address_Of.Set_PTA_Object (...);
5    end if;
6    ...
7    if Node.all in Siml.Routine_Call_T'Class then
8       Node.As_Routine_Call.Set_PTA_Object (...);
9    end if;
10   raise Standard.IML_Roots.Interface_Not_Implemented with "received "
        & Node.Skill_Name.all & " in Value.PTA_Object";
11 end Set_PTA_Object;
```

Quellcode 19: Der ursprünglich generierte Code der Set-Prozedur wird immer eine Ausnahme auslösen.

4.11.2. Fehlende Body Implementierungen

Eine häufige Fehlerursache sind nicht implementierte Unterprogramme, da die Funktionalität bisher nicht benötigt wurde. In diesen Fällen generierte der Codegenerator lediglich das Interface des Unterprogramms. Das pta_tool benötigt diese Unterprogramme, weshalb zur Laufzeit die Ausnahme ausgelöst wird. Quellcode 20 zeigt dies anhand der Funktion Get_Tail_Of_Items aus der Datei *sequences.adb*.

Zur Lösung dieser Probleme können die generierten Unterprogramme der IML-Version analysiert werden. Eine Nachimplementierung mithilfe der in der *SKilL*-Version verwendeten Datentypen war in allen vorgefundenen Fällen möglich. Quellcode 21 veranschaulicht die Implementierung des Beispiels aus Quellcode 20. Eine Besonderheit ist

```
function Get_Tail_Of_Items
 (Node : access Sequence_Class'Class)
  return Standard.Value_Internals.Value is
begin
    raise Standard.Program_Error;
    return Get_Tail_Of_Items (Node);  --  NOT USED BY cafe++
end Get_Tail_Of_Items;
```

Quellcode 20: Das Codebeispiel zeigt die generierte Funktion Get_Tail_Of_Items. Zur Laufzeit wird eine Exception ausgelöst.

hier die Interpretation des *Tails* einer Liste. Die ursprüngliche Implementierung versteht darunter das letzte Element einer Liste und nicht die Liste ohne das erste Element.

```
...
begin
  if (Node.Get_Items.This.Length > 0) then
     return Node.Get_Items.This.Element(Node.Get_Items.This.Length -
         1).Dynamic_Value;
  else
     raise Standard.Program_Error;
     return Get_Tail_Of_Items (Node);  --  NOT USED BY cafe++
  end if;
end Get_Tail_Of_Items;
```

Quellcode 21: Der Codegenerator wurde erweitert, sodass die Funktion Get_Tail_Of_Items aus Quellcode 20 das letzte Element der Node Liste zurück gibt. Dies entspricht der semantischen Implementierung der Get_Tail_Of_Items Funktion der IML-Version.

Viele weitere Funktionen konnten nach gleichem Vorgehen generiert werden.

4.11.3. Fehlende Copy Implementierung für generierte Sets

Felder von generierten Sets besitzen eine Kopierfunktion Copy_Feldname. Analog zu dem beschriebenen Fehler in Kapitel 4.11.2 ist auch bei dieser Funktion der *Body* nicht funktionsfähig generiert worden. Aufgrund einer Besonderheit wird dieser Fehler jedoch explizit geschildert. Quellcode 22 zeigt einen solchen Codeauszug aus der generierten Datei *projects/libs/iml/generated1/units.adb*.

Die dazugehörigen Sets, in diesem Fall *O_Node_Internals*, verfügen ebenfalls über eine Copy Funktion, die zur Lösung dieses Problems herangezogen wird. Copy_Feldname ruft explizit die Kopierfunktion des entsprechenden Sets auf und fungiert somit als eine Wrapperfunktion. Die notwendige Implementierung im Codegenerator ist in Quellcode 23 abgebildet.

```
1 function Copy_Declaration_Table
2  (Node : access Unit_Class'Class)
3   return Standard.O_Node_Internals.Sets.Set is
4 begin
5   raise Standard.Program_Error;
6   return Copy_Declaration_Table (Node);  --  NOT USED BY cafe++
7 end Copy_Declaration_Table;
```

Quellcode 22: Exemplarischer Auszug einer generierten Copy Funktion für Sets. In diesem Fall handelt es sich um die Funktion *Copy_Declaration_Table* aus dem Paket *projects/libs/iml/generated1/units.adb*.

```
1 function Copy_${name(f)}
2  (Node : access ${name(t)}_Class'Class)
3   return ${mapType(b).replaceFirst(s"\\.${name(b)}$$", "")}.Sets.Set
       is
4 begin
5 """)
6   if ("Begin_Objects".equals(name(f)) || "End_Objects".equals(name(f)
       )) {
7   out.write(s"""
8   raise Standard.Program_Error;
9   return Copy_${name(f)} (Node);  --  NOT USED BY cafe++
10  """)}
11 else {
12  out.write(s"""
13  return ${mapType(b).replaceFirst(s"\\.${name(b)}$$", "")}.Sets.Copy
       (Node.Get_${name(f)});
14  """)}
15 out.write(s"""
16 end Copy_${name(f)};
```

Quellcode 23: Erweiterung des Codegenerators mit dem benötigten Copy Aufruf. Änderungen vorgenommen in der Datei *imlgen4/src/main/scala/codegen/BodyMaker.scala*.

In Zeile 6 wird zusätzlich geprüft, ob es sich um die Felder *Begin_Object* oder *End_Object* handelt. Beide besitzen zurzeit noch keine Getter-Methoden, weshalb der genutzte Ansatz fehlschlägt. Aus diesem Grund wird für beide Felder weiterhin die Ausnahme generiert. Auf alle verbleibende Felder kann mit der Getter-Methode zugegriffen werden. Dieses wird als Parameter mit einem expliziten Funktionsaufruf zur Copy Methode übergeben. Der so generierte Code ist Quellcode 24 in dargestellt.

```
1 function Copy_Declaration_Table
2  (Node : access Unit_Class'Class)
3   return Standard.O_Node_Internals.Sets.Set is
4 begin
5   return Standard.O_Node_Internals.Sets.Copy(Node.
        Get_Declaration_Table);
6 end Copy_Declaration_Table;
```

Quellcode 24: Exemplarische Copy_Feldname funktion aus dem Beispiel Quellcode 22. Nun mit implementierten *Body* basierend auf dem Codegeneratorauszug in Quellcode 23.

4.12. Vereinfachungen

An vereinzelten Stellen im Programm ermöglichte die Migration bestehende Implementierungen mit neuen *SKilL*-Funktionen zu vereinfachen, ohne die semantische Bedeutung zu gefährden.

Dieses Kapitel gibt einen Einblick, welche Vereinfachungen möglich sind und typischerweise angewandt wurden.

4.12.1. Vereinfachung mit neuer Append Funktion

Die in Kapitel 4.7.1 und Kapitel 4.7.2 vorgestellten *SKilL*-Container stellen eine neue *Append* Funktion zur Verfügung. Mit dieser können verschiedene Codekonstrukte, insbesondere bei der Initialisierung, lesbarer gestaltet werden. Zur Veranschaulichung wird in Quellcode 25 das Beispiel der Prozedur *Add_Callee* aus dem Paket *Thread_DF_Patterns* herangezogen. Die Prozedur fügt ein Element in ein Array ein. Existiert das Array noch nicht, wird es erstellt. Ist das Array bereits vorhanden, wird ein neues temporäres Array mit einem zusätzlichen Element allokiert und alle Elemente des originalen Arrays in das neue Array kopiert. An letzter Position wird das neue Element eingefügt. Die exakte Implementierung ist in Quellcode 25 dargestellt.

Verwendet man nun die *Append* Funktion des *SKilL-Containers-Vectors*, kann dieselbe Implementierung deutlich lesbarer gestaltet werden. Quellcode 26 zeigt die überarbeitete Version derselben Prozedur.

4.12.2. Ada 2012 Iterator

Ada 2012 führte die *Generalized Loop Iteration* [ISO/IEC, 2012, Kapitel 5.5.2], [Barnes, 2014, Kapitel 21.6], bei welcher sehr übersichtlich über ein Array iteriert werden kann, ein. Die verwendete Kontrollvariable ist anstelle des üblichen Indexes das aktuelle Array Element. Das Auslesen des Elementes und zusätzliche lokale Variablen können auf diese Weise eingespart werden, der Code wird lesbarer. Modifikationen von Iterator Code Passagen, wie zum Beispiel in Kapitel 4.7.3 beschrieben, wurden üblicherweise mit Hilfe der *Generalized Loop Iteration* aufgelöst. Der Vergleich in Quellcode 27

```
1  procedure Add_Callee
2   (Pattern : access Thread_DF_Pattern_Class;
3    Site    : in     Routine_Calls.Routine_Call;
4    Callee  : in     Thread_DF_Pattern)
5  is
6    Site_Info : Call_Site_Info renames Pattern.Call_Sites.Elements
7      (Routine_Calls.Get_Site_Id (Site));
8    Original  : Pattern_Array_Access := Site_Info.Callees;
9    use type Routine_Calls.Routine_Call;
10 begin
11   pragma Assert (Site_Info.Site = Site);
12   if Original /= null then
13      Site_Info.Callees :=
14        new Pattern_Array (Original'First .. Original'Last + 1);
15      Site_Info.Callees (Original'Range) := Original.all;
16      Free (Original);
17   else
18      Site_Info.Callees := new Pattern_Array (1 .. 1);
19   end if;
20   Site_Info.Callees (Site_Info.Callees'Last) := Callee;
21 end Add_Callee;
```

Quellcode 25: Ursprüngliche Prozedur Add_Calle aus Paket Thread_DF_Pattern

```
1  procedure Add_Callee
2   (Pattern : access Thread_DF_Pattern_Class;
3    Site    : in     Routine_Calls.Routine_Call;
4    Callee  : in     Thread_DF_Pattern)
5  is
6    Site_Info : Siml.Call_Site_Info_Dyn renames Pattern.Get_Call_Sites.
       This.Element
7      (Routine_Calls.Get_Site_Id (Site));
8    Original  : Siml.Skill_Array_threaddfpattern.Ref := Site_Info.
       Get_Callees;
9  begin
10   pragma Assert (Site_Info.Get_Site.Dynamic_Routine_Call = Site.
       Dynamic_Routine_Call);
11   if Original = null then
12      Site_Info.Set_Callees (Siml.Skill_Array_threaddfpattern.Make);
13   end if;
14   Site_Info.Get_Callees.This.Append (Callee.As_Thread_DF_Pattern);
15 end Add_Callee;
```

Quellcode 26: Migrierte Prozedur Add_Calle aus Paket Thread_DF_Pattern

zeigt den resultierenden Vorteil. Es werden zwei lokale Variablen und drei Codezeilen weniger benötigt.

```
      package body DF_Patterns is
         Caller_Pattern  : access DF_Pattern_Class)
        return Siml.Skill_Set_v64.Ref
      is
-        Iterator : Locators.Sets.Iterator;
-        Locator  : Locators.Locator;
         Tmp      : Siml.Skill_Set_v64.Ref;
         Result   : Siml.Skill_Set_v64.Ref;
      begin
         Result := Siml.Skill_Set_v64.Make;
-        Iterator := Locators.Sets.Make_Iterator (Callee_Locators);
-        while Locators.Sets.More (Iterator) loop
-           Locators.Sets.Next (Iterator, Locator);
+        for Locator of Callee_Locators.This loop
            Tmp := PTA_DF_Patterns.May_Map_To_Caller
-             (Locator,
+             (Integer (Locator),
               Callee_Pattern.As_PTA_DF_Pattern,
               Routine_Call,
               ...
         end loop;
-        Locators.Sets.Destroy (Iterator);
         return Result;
      end May_Map_To_Caller;
```

Quellcode 27: Auflösen der Funktion Make_Iterator mit Hilfe der *Generalized Loop Iteration* im Paket *DF_Pattern*

4.13. Entfernte Dateien und Funktionen

Im Kontext dieser Arbeit wurden teilweise Dateien und Funktionen aus dem Bauhaus-Projekt ausgeklammert. Hierzu sind die betroffenen Dateien ein einen *src-unused* Ordner verschoben worden. Diese Maßnahmen ermöglichten die Fertigstellung eines ersten migrierten Standes im Rahmen dieser Arbeit. Weiterhin wird auf diese Weise die Wartbarkeit des Systems erhöht. Das folgende Kapitel enthält eine Übersicht dieser Änderungen.

4.13.1. Andersen und Das Werkzeuge

In Kapitel 7.1.1 werden die Abhängigkeiten des pta_tools beschrieben. Über die Bibliothek lib_ptaquery existiert eine Schnittstelle zu den Werkzeugen andersen, das und ecr. Diese Schnittstelle ist einzig in der Datei *pta_query_factory.adb* implementiert. Die Verknüpfung zum pta_tool erfolgt einzig in der Datei *pta_query_factory.adb* aus der Bibliothek lib_ptaquery. Um den Migrationsaufwand für das pta_tool zu reduzieren, sind die Komponenten dieser beiden Werkzeuge temporär aus dem pta_tool-Build herausgenommen. Analog zu dem Vorgehen in [Przytarski, 2016] sind die entsprechenden Quelldateien in separate *src-unused* Ordner verschoben worden. Die betroffenen Pakete sind im Anhang A vollständig aufgelistet. Zusätzlich wurde die Codeverknüpfung in Quellcode 28 auskommentiert. Sobald die beiden Werkzeuge andersen und das compi-

lierbar sind, kann diese Änderung rückgängig gemacht werden.

```
if Match (Tool_Name, "ecr") then
    Query_Instance := ECR_Interface.Make_Query (IML_Graph, Info);
-- temporarly disable call to das:
-- elsif Match (Tool_Name, "das") then
    -- Query_Instance := DAS_Interface.Make_Query (IML_Graph, Info);
-- temporarly disable call to andersen:
-- elsif Match (Tool_Name, "andersen") then
    --Query_Instance := Andersen_Interface.Make_Query (IML_Graph,
        Info);
else
    raise Unknown_Points_To_Analysis;
end if;
```

Quellcode 28: Auskommentieren der Verbindung zu den Werkzeugen Andersen und Das

4.13.2. PSI_Handling

PSI_Handling ist nach mündlicher Absprache mit Timm Felden als nicht funktionsfähig eingestuft worden. Das Werkzeug ist im originalen Bauhaus nicht compilierbar und wird auch nicht verwendet. Abhängigkeiten zum Paket *PSI_Handling* können somit im ersten Migrationsschritt dieser Arbeit ausgelassen werden. Tabelle 4 listet alle Prozeduren auf, die im Rahmen dieser strategischen Entscheidung aus aktiven Paketen gelöscht wurden.

Prozedur	Paket	Aufrufe
Inserted_Value_Before	*Thread_DF_Patterns*	ausschließlich in *PSI_Handling*
Inserted_Value_After	*Thread_DF_Patterns*	ausschließlich in *PSI_Handling*

Tabelle 4: Gelöschte *PSI_Handling* Funktionen

Im Anhang A sind alle Dateien gelistet, die im Rahmen dieser Vereinfachung nicht mehr Teil des Buildprozesses sind.

4.13.3. Simple_DF_Patterns

Wie bereits in Kapitel 4.6 angemerkt, ist das Paket *Simple_DF_Patterns* ebenfalls nicht funktionsfähig. Um keinen toten Code zu migrieren, wurde die Komponente vollständig aus der libiml entfernt.

4.13.4. Alte IML-Unterprogramme

Der IML-Graph benötigt zur Initialisierung verschiedene Prozeduren, die einen ordnungsgemäßen Zugriff auf die einzelnen Knoten ermöglichen. Konkret handelt es sich um:

- `procedure Load_Node`
- `procedure Save_Node`
- `procedure Mark_Node`

Der *SKilL*-Graph ist auf diese Unterprogramme nicht mehr angewiesen. Aus diesem Grund können die genannten Prozeduren im Rahmen der Migration entfernt werden.

4.14. Semantische Abweichungen

An verschiedenen Stellen der Arbeit kommt es zu semantischen Abweichungen gegenüber der IML-Version. Aus strategischen Gründen wurden Annahmen getroffen, die die Migration im ersten Schritt vereinfachen und nicht zwangsläufig den originalen IML-Code widerspiegeln. Die Idee war es, einen ersten compilierbaren Stand zu erzeugen, der als Ausgangspunkt für alle weiteren Migrationsschritte genutzt werden kann. In Kapitel 4.14.1 und Kapitel 4.14.2 sind solche Entscheidungen beschrieben. An anderen Stellen im Code war der ursprüngliche Implementierungsgedanke nicht immer nachvollziehbar. Es war nicht immer klar, ob der vorliegende Code in dieser Form notwendig ist und ob es tatsächlich einen entsprechenden Anwendungsfall gibt. Das Codebeispiel in Kapitel 4.14.3 repräsentiert einen solchen Fall.

4.14.1. Fehlende Sortierung

Das in Kapitel 4.7.3 portierte *Ordered_Sets* verwaltet seine Elemente mithilfe eines AVL-Baumes, weshalb die Elemente des Sets sortiert sind. Auch das in Kapitel 4.7.2 beschriebene *Resizable_Arrays* kann in sortierter Form vorliegen. Die verwendeten *SKilL*-Container sind hingegen nicht sortiert. Es ist nicht bekannt, ob die Sortierung für das korrekte Programmverhalten erforderlich ist oder ob die Datentypen aus anderen Gründen, wie zum Beispiel besserer Performanz, gewählt worden sind.

An einigen Stellen im Programm wurde die Funktion Is_Sorted aus dem Paket *Resizable_Array* nachimplementiert. Die Funktion wird in Assert-Überprüfungen verwendet um sicherzustellen, dass ein *Resizable_Array* tatsächlich sortiert ist. Bei den durchgeführten Laufzeittest ist keine dieser Assert Überprüfungen bei den überprüften *SKilL-Containers-Vectors* ausgelöst worden. Weiterhin wurden auch keine negative Einflüsse auf die Performanz beobachtet. Eine funktionale Validierung steht noch aus, siehe hierzu auch Kapitel 7.3.

4.14.2. Fehlende Less-Implementierung für die generierten Typen

In den Übersetzungseinheiten des pta_tools wurde von einzelnen generierten Typen eine *Less* Funktion erwartet. Anstatt eine solche Funktionalität lokal zu implementieren, wurde im Rahmen dieser Arbeit der Codegenerator codegen angepasst, sodass die *Less*-Implementierung allen generierten Typen zur Verfügung steht.

```
1  with Ada.Containers;
2  use type Ada.Containers.Hash_Type;
3
4    function Less
5      (X : in ${name(t)};
6       Y : in ${name(t)})
7       return Standard.Boolean is
8       (siml.Hash(siml.To_${name(t)}(X.To_Annotation)) < siml.Hash(
           siml.To_${name(t)}(Y.To_Annotation)));
```

Quellcode 29: Implementierte *Less* Funktion in Codegenerator codegen in der Datei imlgen4/src/main/scala/codegen/SpecMaker.scala

Die Ordnung kann dabei sehr einfach mithilfe der Hash-Funktion, die jedem *SKilL*-Typ zugeordnet ist, bestimmt werden. Dies ist möglich, da die zugeordneten Hashwerte mit den eigentlichen Eingabewerten übereinstimmen (modulo 2^{32}). Quellcode 29 zeigt die so implementierte Erweiterung. Diese Implementierung basiert auf der Annahme, dass keine Werte größer 2^{32} entstehen. Die Vereinfachung ist dadurch begründet, dass in der bisherigen IML-Implementierung ausschließlich mit *Integer_32* gerechnet wird. Die gewählte Umsetzung ist aus Zeitgründen für diese Arbeit ausreichend, langfristig sollte an dieser Stelle jedoch eine exaktere Lösung angestrebt werden.

4.14.3. Abgeänderte Copy Prozedur aus Resizable_Array

Quellcode 30 zeigt den IML-Code der Copy Prozedur von *Resizable_Array*. Die Bedingung in Zeile 13 scheint fragwürdig. Wenn sowohl *Source* als auch *Target* auf dasselbe Array zeigen, wird ein neues Array für das *Target* allokiert und das Array somit geklont. Ist der Startindex beider Arrays identisch, wird das *Target* Array modifiziert, sodass die Kapazitäten beider Arrays übereinstimmen. Das ursprüngliche *Target* Array bleibt aber in diesem Fall erhalten. In den restlichen Fällen wird das *Target* Array deallokiert und ein neues Feld mit den *Source* Eigenschaften allokiert.

Quellcode 31 zeigt einen exemplarischen Aufruf dieser Copy Prozedur. *Source.Derefs* und *Target.Derefs* sind dabei Zeiger auf *Resizable_Arrays*. Die Funktion ruft die Copy Prozedur des *Resizable_Arrays* auf.

```
procedure Copy
 (Source : access constant Resizable_Array;
  Target : in out          Pointer)
is
begin
  if Target = null then
     if Source = null then
        null;
     else
        Target := New_Array (First => Real_First (Source.all),
                             Initial_Last => Last (Source));
     end if;
  elsif Dummy_Pointer (Target) = Dummy_Pointer (Source) then
     Target := New_Array (First => Real_First (Source.all),
                          Initial_Last => Last (Source));
  elsif First (Source) = First (Target) then
     Set_Range (Target, Last (Source));
  else
     Free (Target);
     Target := New_Array (First => Real_First (Source.all),
                          Initial_Last => Last (Source));
  end if;
  for I in First (Target) .. Last (Target) loop
     Target.Elements (I) := Source.Elements (I);
  end loop;
end Copy;
```

Quellcode 30: Copy Prozedur aus IML *Resizable_Array*.adb

```
procedure Copy
 (Source : in     Dereference_Set;
  Target : in out Dereference_Set)
is
begin
  Deref_Pair_Arrays.Copy
    (Source => Source.Derefs,
     Target => Target.Derefs);
end Copy;
```

Quellcode 31: Exemplarischer Aufruf Copy Funktion für SKILL_Container_Arrays

Quellcode 32 veranschaulicht die vereinfachte *SKilL*-Version des Aufrufs. Sie ersetzt die Prozedur in Quellcode 31.

Die neue Prozedur löscht zunächst alle Elemente des *Targets* und kopiert anschließend jedes Element von *Source* nach *Target*. Sollte ein oben beschriebener Sonderfall auftreten, wird zur Laufzeit eine Ausnahme geworfen. Die Funktion sollte dann entsprechend erweitert werden. Während den Laufzeittest wurde diese Ausnahme nicht ausgelöst.

```
1  procedure Copy
2   (Source : in     Dereference_Set;
3    Target : in out Dereference_Set)
4  is
5    use Siml;
6  begin
7    if Source.Derefs = Target.Derefs then
8       raise Program_Error with "Error because simplification during
            migration from IML to SKILL";
9    elsif Target.Derefs = null then
10      raise Program_Error with "Error because simplification during
            migration from IML to SKILL";
11   end if;
12   Target.Derefs.This.Clear;
13   for I in 0 .. Source.Derefs.This.Length - 1 loop
14      Target.Derefs.This.Append (Source.Derefs.This.Element (I));
15   end loop;
16 end Copy;
```

Quellcode 32: Vereinfachte Copy Funktion für SKILL_Container_Arrays

5. Laufzeitfehler

Nach der erfolgreichen Migration des pta_tools konnte dieses erstmals compiliert und ausgeführt werden. Dieses Kapitel beschreibt sämtliche Fehler, die im Rahmen der Ausführung des pta_tools aufgetreten sind und nicht in Kapitel 4 behandelt wurden.

In aller Regel handelt es sich um Laufzeitausnahmen, die vom Ada-Laufzeitsystem ausgelöst werden. Vereinzelt wurden jedoch auch Fehler behoben, die keine Ausnahme ausgelöst haben, stattdessen aber zu ungewollten Programmverhalten geführt hätten (bspw. Kapitel 5.1).

5.1. Propagierung falscher Indizes

In Kapitel 4.7.1 wurde bereits die Problematik der Index-Verschiebung im Rahmen der Migration von Ada-Arrays bzw. *Resizable_Array* hin zu *SKilL-Containers-Vectors* erläutert. Alle analysierten *Resizable_Array* starten mit dem Index 1, die *SKilL* Gegenstücke (*SKilL-Containers-Vectors*) jedoch mit dem Index '0'. Verwendet man Funktionen wie *First* oder *Last* aus Kapitel 4.7.2, ist dies bei der Migration zu berücksichtigen. Problematisch wird es jedoch, wenn mit harten Indizes auf verschiedene Elemente zugegriffen wird. Dann müsste die *„off by one“*-Verschiebung berücksichtigt werden. Quellcode 33 zeigt einen vorgefundenen Codeausschnitt im Paket PTA_Tool.adb. Die While-Schleife iteriert ursprünglich über ein komplettes *Resizable_Array*. Im Migrationsprozess muss folglich die Variable *Object* mit '0' initialisiert bzw. die angepasste Funktion *First* verwendet werden. Ansonsten wird in diesem Fall das erste Element des *SKilL*-Vectors nicht gesetzt.

```
1  declare
2      Object : PTA_Objects.PTA_Object;
3  begin
4      Object := 1;
5   while Object <= PTA_Objects.Last (Map => PTA_Sets) loop
6      ...
7      PTA_Objects.Set_PTA_Set (..., Index => Object, ...);
8      ...
9      Object := PTA_Objects.PTA_Object'Succ (Object);
10  end loo;
11 end;
```

Quellcode 33: Auszug aus der Funktion Assign_Pointsto_Set in pta_tool.adb

Das Problem an diesem Beispiel ist, dass der gesamte Code ohne Anpassungen compiliert. Der Typ der Variable PTA_Sets wurde zwar geändert, die Änderungen verursachten aber an dieser Stelle keinen Fehler, sondern lediglich in der ausgelagerten Funktion *.Last*. Das bedeutet, dass der Codeausschnitt nicht bearbeitet werden musste, was wiederum die Chance einen solchen Fehler zu finden, deutlich minimiert.

5.2. Veränderte Default-Werte

In Kapitel 4.1 wurden bereits mehrere Compilerfehler, welche durch das Auslagern der Datentypen hervorgerufen werden, geschildert. Zur Laufzeit offenbarte sich ein Effekt hinsichtlich der Default-Initialisierung der Objekte. Viele ursprüngliche IML-Datentypen bekommen bei ihrer Definition Default-Werte zugewiesen. Die in der *SKilL*-Version neu eingefügten Untertypen sind Zeiger auf *SKilL*-Objekte und der Default-Wert eines nicht initialisierten Access Typs ist *Null*. Eine Laufzeitprüfung auf den Default-Wert schlägt folglich fehl und der Programmfluss wird sich verändern.

Quellcode 34 veranschaulicht das Problem. Es wird ein Record definiert und mit Default-Werten initialisiert. Weiterhin wird eine Variable *No_PTA_Ref* angelegt, die den Initialisierungsfall darstellt.

```
1  type PTA_Ref is
2      record
3         Kind   : Ref_Kind := Unknown;
4         Part   : PTA_Objects.PTA_Object := 0;
5         Offset : PTA_Objects.PTA_Offset := 0;
6         Width  : PTA_Objects.PTA_Length := 0;
7      end record;
8
9  No_PTA_Ref : constant PTA_Ref :=
10          (Kind => Unknown, Part => 0, Offset => 0, Width => 0);
```

Quellcode 34: Ursprüngliche Definition des Datentyps PTA_Ref im Paket PTA_Objects.ads

Quellcode 35 zeigt die migrierte Version des vorgestellten Beispiels. *PTA_Ref* ist ein Untertyp auf einen Zeiger, der auf ein hierarchisches Objekt zeigt. *No_PTA_Ref* allokiert eine Variable, die dem ursprünglichen Default-Fall entspricht.

```
1  subtype PTA_Ref is Siml.PTA_Ref_Dyn;
2
3  No_PTA_Ref : constant PTA_Ref :=
4      Skill_State.Get_Constructor_Graph.PTA_Refs.Make
5          (F_Kind   => Interfaces.Integer_64 (Ref_Kind'Pos (Unknown)),
6          F_Offset => 0,
7          F_Part   => 0,
8          F_Width  => 0).Dynamic_PTA_Ref;
```

Quellcode 35: Migrierte Version von Quellcode 34 im Paket PTA_Objects.ads

Das Problem ist die Abfrage in Quellcode 36. Ein nicht initialisierter Access Typ *PTA_Ref* besitzt als Default-Wert *Null*. Eine Abfrage auf *No_PTA_Ref* wird somit immer scheitern, die *SKilL*-Version des Codes führt den ELSE-Zweig des Beispiels aus. Es ist schwierig, dies zur Laufzeit zu entdecken, außer der ausführbare Code greift auf die *No_PTA_Ref* Variable zu. In diesem Fall wird das Ada-Laufzeitsystem eine Aus-

nahme auslösen. Wird nicht auf die Variable zugegriffen, so wird auch keine Ausnahme geworfen und dieser Effekt bleibt bei einfachen Laufzeittests unentdeckt.

```
1 declare
2     Foo: PTA_Ref;
3 begin
4     if Foo = No_PTA_Ref then
5         ...
6     else
7         ...
8     end if;
9 end;
```

Quellcode 36: Pseudocode zur Abfrage auf den Initialwert

Die schnellste Lösung für dieses Problem ist es, bei der Abfrage zusätzlich mit einem ELSE-Statement auf *Null* zu prüfen. Dies müsste aber bei jeder entsprechenden (und auch zukünftigen) Prüfung eingebaut werden. Ein hoher Wartungsaufwand und eine große Fehleranfälligkeit sind die Folge.

Alternativ könnte man *No_PTA_Ref* ebenfalls mit *Null* initialisieren, um einen identischen Kontrollfluss zu erhalten. Dies würde aber eventuelle Programmkonstrukte zerstören, die mithilfe von *No_PTA_Ref* gezielt Variablen vom Typ *PTA_Ref* auf das Tupel (Unknown, 0, 0, 0) setzen wollen. Anschließende Zugriffe auf die einzelnen Felder würden zur Laufzeit scheitern. Weiterhin erzeugt dies gewissermaßen das umgekehrte Problem. Prüfungen auf das Tupel bzw. in der *SKilL*-Version auf ein *PTA_Ref* Objekt, dessen Felder genau mit diesen Werten initialisiert sind, werden scheitern.

Im Kontext dieser Masterarbeit wurden beide Ansätze getestet. Aufgrund der besseren Testergebnisse wurde sich für den ersten Ansatz entschieden.

5.3. Nicht vollständig initialisierte Resizable_Arrays

In Kapitel 4.7.2 ist die Konvertierung der *Resizable_Array* Prozedur Set_Range beschrieben. Mithilfe dieser kann die Kapazität des *Resizable_Arrays* gegebenenfalls erhöht werden. Als *SKilL*-Pendant wird die Prozedur Ensure_Allocation von *SKilL-Containers-Vectors* verwendet. Diese initialisiert neu allokierte Elemente jedoch mit *Null*, weshalb zur Laufzeit eine Ausnahme ausgelöst werden kann. Quellcode 37 zeigt einen entsprechenden Quellcodeausschnitt. Die FOR-Schleife stellt nachträglich sicher, dass alle *Null* Elemente des *SKilL-Containers-Vectors* initialisiert sind.

```
1 ...
2 Result.Get_Infos.This.Ensure_Allocation (Integer (Last_Object));
3 for I in 0 .. Result.Get_Infos.This.Length - 1 loop
4  if Result.Get_Infos.This.Element (I) = null then
5    ObjectInfo := Skill_State.Get_Constructor_Graph.Object_Infos.Make;
6    Result.Get_Infos.This.Replace_Element (I, ObjectInfo);
7  end if;
8 end loop;
```

Quellcode 37: Der *SKilL-Containers-Vectors*, auf welchen mit dem Getter *Result.Get_Infos* zugegriffen wird, wird nach der Verwendung von Ensure_Allocation mit einer FOR-Schleife initialisiert.

5.4. Nicht implementierte Unterprogramme

Verschiedene generierte Unterprogramme sind noch nicht vollständig implementiert, weshalb ihre *Bodys* entsprechende Ausnahmen beinhalten. Bei den Laufzeittests des pta_tools wurden mindestens die folgenden noch nicht implementierten Funktionen aus den Paketen *projects/libs/iml/generated1/units.adb* und *projects/libs/iml/generated1/sequences.adb* benötigt.

- *Copy_Declaration_Table*
- *Copy_Provided_Definitions*
- *Get_Head_Of_Items*
- *Get_Tail_Of_Items*
- *Deallocate*

Da es sich hier zusätzlich zu einem Laufzeitfehler auch um eine Anpassung am Codegenerator codegen handelt, wurde dieses Problem bereits in Kapitel 4.11 im Abschnitt 4.11.3 diskutiert.

5.5. Assert Checks in C-Files

Teile des pta_tools sind in der Programmiersprache C geschrieben. Die entsprechenden Funktionen werden aus den Ada Paketen aufgerufen und ausgeführt. Es ist bisher unklar, aus welchem Grund diese Auslagerung in C erfolgte.

Bei den Laufzeittests schlug in der C-Datei *libs/flowinsens_pta/src/efficientset.c* mindestens ein Assert-Test fehl. Die Ursache des Fehlers konnte nicht direkt nachvollzogen werden, weshalb die Assert-Tests in dieser Datei vorübergehend ausgeschaltet wurden. Hierzu ist im dazugehörigen Headerfile das Macro NDEBUG definiert. Auf diese Weise konnten nachfolgende Fehler behoben werden. Ohne das Macro NDEBUG scheitern derzeit noch ca. 60 % der im nächsten Kapitel beschriebenen Testfälle.

6. Testsuite

Kapitel 6 gliedert sich in zwei Unterkapitel. In Kapitel 6.1 wird der geplante und endgültige Testaufbau der Data Race Werkzeuge beschrieben. Dieser Ansatz ist noch nicht vollständig umgesetzt. Daher beschreibt Kapitel 6.2 den zum Stand dieser Arbeit durchgeführten Testprozess.

6.1. Konzept der finalen Testsuite

Abbildung 6.1 zeigt den prinzipiellen Ablauf der Testsuite für migrierte Werkzeuge. Dieser Ansatz wurde bereits in der Arbeit [Przytarski, 2016] für die Tools imlmetrics und imlstat angewendet.

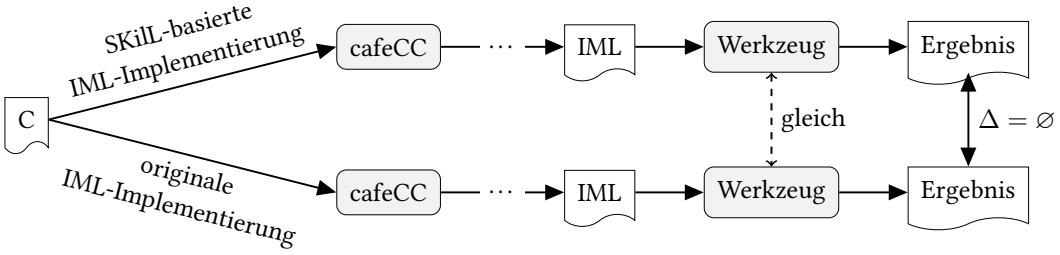

Abbildung 6.1: Ablauf der Testsuite für migrierte Werkzeuge, entnommen aus [Przytarski, 2016, Kapitel 7.1]

Die Validierung der migrierten Werkzeuge erfolgt mittels einer Referenzmessung zu den bereits existierenden IML-Werkzeuge. Hierzu werden zwei verschiedene Bauhaus-Umgebungen, je eine IML- und eine *SKilL*-Version, verwendet. Jede .c-Datei durchläuft die Werkzeugkette bis zu dem zu prüfenden Werkzeug. Der Test gilt als bestanden, wenn die Analyseausgabe beider Werkzeugversionen identisch ist. Angenommen die Werkzeuge sind deterministisch und erzeugen ein lesbares Ergebnis, so kann die Validierung beispielsweise mit einem MD5-Summenabgleich der Ausgaben realisiert werden.

Da der Linker für die *SKilL*-Binärdateien noch nicht entwickelt ist, werden in diesem Testszenario nur einzelne .c-Dateien verwendet.

6.2. Aktueller Stand

Das migrierte pta_tool erzeugt als Ergebnis eine IML- bzw. *SKilL*-Binärdatei, die von den Folgewerkzeugen weiter analysiert wird. Der automatische Vergleich beider Binärdateien ist technisch aufwendig. Aus diesem Grund wird beim derzeitigen Migrationsstand einzig ein einfacher Laufzeittest durchgeführt. Die gefundenen Laufzeitfehler sind in Kapitel 5 dokumentiert. Die automatisierte vollständige inhaltliche Prüfung kann einfach umgesetzt werden, sobald das vierte Werkzeug der Data Race Werkzeugkette, raceq, migriert ist und ein lesbares Ergebnis erzeugt wird. In dieser Arbeit sind hingegen indirekte inhaltliche Tests durchgeführt worden. Diese sind in Kapitel 6.5 beschrieben.

Ziel dieser Testsuite ist einzig der Vergleich der beiden pta_tool-Versionen. Im Gegensatz zu dem in Abbildung 6.1 vorgestellten Konzept wird derselbe Compiler für die Generierung der .iml-Dateien verwendet. Auf diese Weise können Abweichungen aufgrund unterschiedlicher cafecc-Versionen ausgeschlossen werden. Weiterhin werden .c-Dateien, für die keine .iml.sf-Dateien generiert werden konnten, von der Teststatistik ausgeschlossen. Abbildung 6.2 veranschaulicht die Pfade der Testdurchgänge. Blaue Knoten symbolisieren Komponenten der *SKilL*-Version, grüne Knoten die der IML-Version.

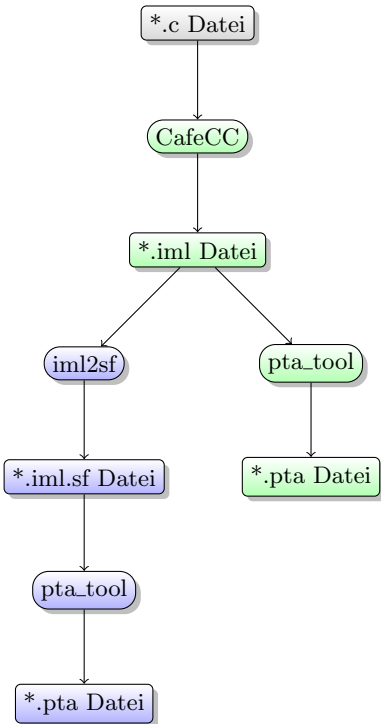

Abbildung 6.2: Derzeitiger Testablauf für das migrierte pta_tool

6.2.1. Testumgebung

Die Tests wurden auf dem Institutsrechner *pslx0* ausgeführt.

Betriebssystem	Debian 8.10 (Jessie); linux-image-3.16.0-5-amd64 (3.16.51-3+deb8u1)
Compiler	GNAT Pro 7.2.2 (20140525)
IML-Bauhaus	commit 0d98771336a6ee3442e74db38749fbd8e0bbed8d
***SKilL*-Bauhaus**	commit 462fd18a118f93124b1e8c1de0abe55866c5b615

Die beiden Bauhausumgebungen stammen aus demselben Repository. Die IML-Version ist ein Softwarestand, bevor erste Erweiterungen hinsichtlich *SKilL* vorgenommen wurden.

Die verwendeten Argumente der Werkzeugaufrufe können dem Pseudocode in Quellcode 38 entnommen werden.

```
INFO: Execute CafeCC:
cafeCC foo.c -B 'pwd' -c -o out/foo.iml
INFO: Execute iml2sf:
iml2sf out/foo.iml
INFO: Move skill binary file to out folder:
mv foo.iml.sf out/foo.iml.sf
INFO: Start pta_tool (IML):
pta_tool out/foo.iml -o out/foo.pta_iml -thread_api pthread
INFO: Start pta_tool (SKILL):
pta_tool out/foo.iml.sf -o out/foo.pta_skill -thread_api pthread
```

Quellcode 38: Exemplarischer Aufruf der Werkzeugkette mit der Datei *foo.c*

6.3. Testwerkzeug

Die Testsuite besteht aus vier kleinen Python-Skripten, die einen automatisierten Testablauf ermöglichen. Tabelle 5 listet die zugehörigen Dateien auf.

Dateiname	Aufgabe
fileManager.py	Hilfsklasse zum Arbeiten mit dem Dateisystem
commandLineParser.py	Hilfsklasse zum Verarbeiten der Konsolen Argumente
bauhausController.py	Steuert den Aufruf der Bauhauswerkzeuge
testSuite.py	Testkontroller, steuert den Testablauf

Tabelle 5: Übersicht der Python Klassen zum Ausführen der automatisierten Tests

Die Testapplikation liest alle .c-Dateien in einen Ordner ein und durchläuft den in Abbildung 6.2 beschriebenen Ablauf. Alle gescheiterten Aufrufe der Werkzeuge werden gezählt und am Ende mit der Gesamtzahl der ausgeführten Durchläufe ausgegeben. Die Testsuite beinhaltet ebenfalls einen automatisierten MD5-Summenabgleich, welcher mit dem Werkzeug imlmetrics getestet wurde. Imlmetrics liegt bereits in einer IML- und *SKilL*-Version vor und erzeugt eine lesbare Ausgabe.

6.4. Testdaten

Für das pta_tool existieren noch keine Testdaten, weshalb eigene Testsets ausgewählt wurden. Dabei wurden drei Ansätze verfolgt:

Testset 1: Data Race Beispiele Es wurden vier einfache .c-Dateien implementiert, die einen Data Race Konflikt verursachen und nachweisbar die erwarteten Warnungen des raceq Werkzeuges hervorrufen. Einer der Testfälle simuliert mithilfe von Funktionszeigern gemäß [Keul, 2011] eine *false positive* Warnung. Sobald inhaltliche Tests möglich sind, empfiehlt es sich, diese Kategorie zu erweitern.

Testset 2: Existierende Open Source C-Dateien Um möglichst viele verschiedene C-Programmierkonstrukte mit dem pta_tool zu durchlaufen, ist es ratsam, eine größere Ansammlung von .c-Dateien zu untersuchen. Aus diesem Grund wurde ein Testset aus der Testsuite der GNU Compiler Collection (gcc) ausgewählt.[4] Von dem ausgewählten Testset, *c-c++-common*, konnten aus 494 .c-Dateien 369 .iml.sf-Dateien generiert werden. Die gescheiterten 125 Compiliervorgänge wurden aus Zeitgründen nicht näher untersucht. Zum einen sind sie für diese Arbeit nicht relevant, zum anderen bilden 369 .iml.sf-Dateien eine ausreichend große Basis für erste Laufzeittests.

Testset 3: Testdateien der Vorgängerarbeit Mit dem Ziel, die Testtiefe des Vorgängerprojektes zu erhalten, wurden zusätzlich die Testdateien aus [Przytarski, 2016, Kapitel 7.3] integriert. Auf diese Weise werden auch größere Übersetzungseinheiten durchlaufen. „Bei der größten Datei handelt es sich um das Projekt SQLite, das ein einbettbares, relationales Datenbankmanagementsystem ist. Die dazugehörige Testdatei sqlite3.c ist eine aus mehr als 100 verschiedenen Quelltextdateien verkettete Datei.“ [Przytarski, 2016, Kapitel 7.3]

Eine Auflistung aller Testdateien befindet sich in Anhang B.

6.5. Testergebnisse

Mithilfe der Testsuite sind zahlreiche Laufzeitfehler gefunden und erfolgreich behoben worden. Tabelle 6 zeigt die gemessenen Ergebnisse des in Abbildung 6.2 beschriebenen Testablauf.

	SKilL pta_tool		IML pta_tool	
	bestanden	Gesamtanzahl	bestanden	Gesamtanzahl
Testset 1	4	4	4	4
Testset 2	368	369	368	369
Testset 3	35	36	33	36
Summe	407	409	405	409

Tabelle 6: Übersicht der bestandenen Testfälle der beiden pta_tool Versionen

Bestanden bedeutet, dass das pta_tool ohne Fehlermeldung durchgelaufen ist und erfolgreich eine Binärdatei erstellt hat. Tabelle 7 listet die vier nicht *bestandenen* Testfälle auf. Bei Testset 2 kommt es zusätzlich zu einer Abweichung der Gesamtzahl, da bei einem Testfall keine .iml.sf-Datei generiert werden konnte.

In der *Sqlite3.c* Testdatei schlägt beim Ausführen der *SKilL*-Version des pta_tools eine Assert-Überprüfung fehl. Eine Variable entspricht dabei nicht dem erwarteten Wert. Kommentiert man dieser Überprüfung aus, so scheitert die *SKilL*-Version ebenfalls an der bereits beobachteten Assert-Überprüfung der IML-Version. Der Programmablauf scheint folglich in beiden Versionen identisch zu sein. Die Gründe für die Abweichung

[4] `https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite`, Letzter Commit: 2e7e8f5

Testcase	Testset	*SKilL*-Version	IML-Version
squlite3.c	3	Assert Fehler	späterer Assert Fehler
unqlite.c	3	läuft durch	vermutlich Endlosschleife
ph7.c	3	läuft durch	vermutlich Endlosschleife
Wunused-var-14.c	2	Assert Fehler	Assert Fehler an identischer Stelle

Tabelle 7: Kurzanalyse der Testfälle aus Tabelle 6, in denen in mindestens einer Version keine .pta-Datei erzeugt werden konnte.

der ersten fehlgeschlagenen Assert-Überprüfung konnten noch nicht weiter analysiert werden. Die beiden Dateien *unqlite.c* und *ph7.c* verursachen in der IML-Version wahrscheinlich eine Endlosschleife. Der Testvorgang wurde nach mehreren Stunden abgebrochen. Die *SKilL*-Version generiert bei beiden Dateien problemlos eine .pta-Datei innerhalb weniger Sekunden. Die Gründe für die Abweichung beider Werkzeuge konnten noch nicht näher analysiert werden. *Wunused-var-14* scheitert in beiden Versionen an einer Assert-Überprüfung in derselben Codezeile. In Hinblick auf die Referenzmessung gilt dieser Test folglich als bestanden, obwohl keine .pta-Dateien generiert werden können.

Wie in Kapitel 6.2 bereits erläutert, ist ein vollständiger Vergleich der beiden *SKilL*-und IML-Binärdateien technisch aufwendig. Jedoch geben einfach durchzuführende Tests Aufschluss darüber, ob es sich um funktionstüchtige .pta-Dateien handeln könnte.

Zum einen ist die Größe der generierten .pta-Dateien immer etwas größer als die der eingelesenen .iml.sf-Dateien. Dies lässt darauf schließen, dass neue Analyseinformationen im *SKilL*-Graph gespeichert wurden. Dieselbe Eigenschaft lässt sich auch bei den generierten Dateien des Referenzdurchlaufs beobachten.

Weiterhin wurde das Werkzeug imlmetrics zum Testen der generierten *SKilL*-Binärdateien verwendet. Dabei konnte zu allen .pta-Dateien eine valide imlmetrics Ausgabe erzeugt werden. Dies bestätigt, dass es sich bei den generierten .pta-Dateien ausschließlich um funktionsfähige *SKilL*-Binärdateien handelt. In einem weiteren Test wurden die Analyseergebnisse des imlmetrics Werkzeuges zwischen IML-und *SKilL*-Version verglichen. In beiden Fällen diente die im Testdurchlauf nach Abbildung 6.2 generierten .pta-Dateien als Eingangsdatei für imlmetrics. Verglichen wurden die MD5-Summen der imlmetrics Ausgabedateien. Auf diese Weise kann überprüft werden, ob bereits vorhandene Datenstrukturen im *SKilL*-Graph versehentlich modifiziert wurden. In $92,4\%$ der Fälle war die Ausgabe identisch. Bei 31 von den 405 vergleichbaren .pta-Dateien wurden Unterschiede festgestellt. Neben diesen Abweichungen wird in der *SKilL*-Version bei allen Dateien eine Warnung ausgegeben, dass eine Datenstruktur innerhalb der .pta-Datei nicht gelesen werden kann. Eine weiterführende Analyse mit dem Werkzeug *SKilLView* bestätigt, dass der betroffene Typ *PTA_Ref_Array* tatsächlich korrupt ist. *SKilLView* ist ein Werkzeug, mit welchem Knoten des *SKilL*-Graphes analysiert werden. Stichprobenartige Kontrollen weiterer Strukturen, die vom pta_tool benötigt werden, scheinen jedoch ordnungsgemäß initialisiert zu sein.

Zusammengefasst ergeben die durchgeführten Tests folgende Erkenntnisse. Von den 369 verwendeten .iml.sf-Dateien konnte, verglichen mit der Referenzmessung, in 99.3% der Fälle erfolgreich eine .pta-Datei generiert werden. Weiterhin sind alle generierten .pta-Dateien nachweislich valide *SKilL*-Binärdateien. Eine folgende imlmetrics Referenzmessung war in $92,4\%$ der Fälle erfolgreich. Es lässt sich anhand der Testergebnisse noch nicht vollständig zeigen, dass das pta_tool erfolgreich portiert wurde. Die vielen positiven Testergebnisse stützen jedoch die Annahme, dass die Migration dennoch in ihrem Grundsatz erfolgreich verlaufen ist. Die gemessenen Testabweichungen sind vielmehr auf vereinzelte Migrationsfehler, die im Rahmen dieser Arbeit noch nicht behoben werden konnten, zurückzuführen.

7. Zusammenfassung und Ausblick

In diesem abschließendem Kapitel werden die erreichten Ziele dieser Arbeit resümiert. Hierzu fasst Kapitel 7.1 den Umfang der Arbeit zusammen bevor in Kapitel 7.2 die erreichten Ziele beschrieben werden. Anschließend werden in Kapitel 7.3 die offenen Punkte diskutiert, ehe die Arbeit in Kapitel 7.4 mit einem Ausblick abschließt.

7.1. Umfang der Arbeit

Die Data Race Werkzeuge wurden ausgewählt, da diese zum einen eine überschaubare Anzahl an handgeschriebenen Typdefinitionen der Zwischendarstellung verwenden und zusätzlich direkt auf den .iml.sf-Dateien von cafecc aufsetzen. Auf der anderen Seite besitzen die Werkzeuge jedoch zahlreiche Abhängigkeiten zu weiteren Bauhaus-Komponenten, welche überwiegend noch nicht funktionsfähig waren und ebenfalls migriert werden mussten. Eine detaillierte Beschreibung dieser Aufwände ist in Kapitel 7.1.1 gegeben.

Zusätzlich wurden insbesondere während der Laufzeittests Fehler in bereits existierenden Komponenten von Vorgängerprojekten entdeckt. Komponenten, die bis dahin noch nicht genutzt wurden. Aufgrund dessen kann man die Portierung dieses Werkzeuges auch als ein Systemtest der Vorgängerprojekte betrachtet werden. Solche Fehlerfälle sind in Kapitel 7.1.2 aufgelistet.

7.1.1. Abhängigkeiten des pta_tools

Abbildung 7.1 zeigt die Build-Abhängigkeiten des pta_tools basierend auf den .gpr-Files.

Zum einen wird deutlich, dass das pta_tool eine Vielzahl von Abhängigkeiten zu weiteren Komponenten beinhaltet. Die grünen Komponenten werden von nahezu jeder Übersetzungseinheit im Bauhaus-Projekt verwendet. So besitzt auch in dieser Abbildung nahezu jeder Knoten eine Abhängigkeit zu lib_reuse und libiml. Aus Gründen der Übersichtlichkeit sind diese Abhängigkeiten in der Abbildung jedoch nicht dargestellt. Die blauen Einheiten sind vollständig abgebildet und waren schon vor dem Projektstart compilierfähig. Die roten Komponenten konnten schon vor dem Migrationsprozess nicht übersetzt werden.

Für einen erfolgreichen Build des pta_tools bedeutet dies, dass alle roten Komponenten migriert werden müssen. Die in dieser Arbeit migrierte Bibliothek lib_concurrency ist mit 26 MByte, nach libiml, die mit Abstand zweitgrößte Werkzeugbibliothek im derzeitigen *SKilL*-Bauhaus-Build. Die drittgrößte Bibliothek lib_clonedetection ist im Vergleich dazu lediglich 14 MByte groß. Alle restlichen compilierbaren Werkzeugbibliotheken sind durchschnittlich kleiner als 5 MByte. Diese Angaben machen deutlich, dass ein großer Teil der verfügbaren Zeit in die Migration der Abhängigkeiten floss. Weiterhin ist zu bemerken, dass die Bibliotheken auch von zahlreichen weiteren Werkzeugen und Bauhaus-Komponenten verwendet werden. So lassen sich in den Ada Projektdateien 82

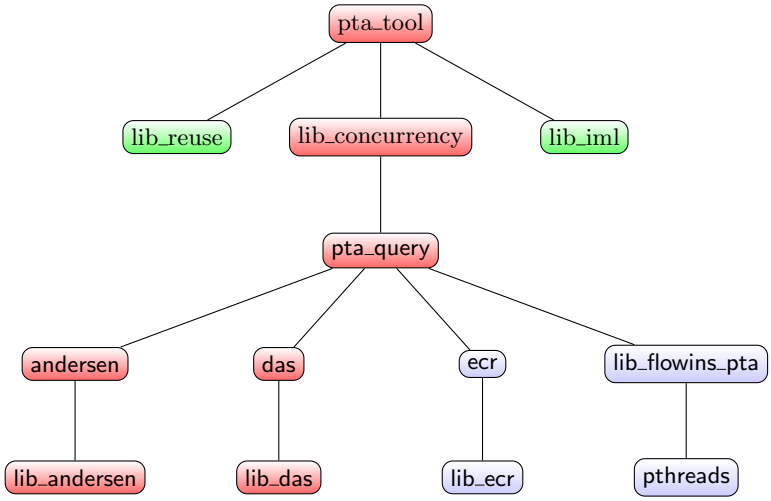

Abbildung 7.1: Abhängigkeitsbaum des pta_tools

Einbindungen der libiml und 14 lib_concurrency Aufrufe zählen. Da die lib_concurrency bis dato nicht compilierfähig war, ist anzunehmen, dass alle 14 Nutzer ebenfalls nicht funktionsfähig sind. Dies führt zur Vermutung, dass ein Großteil des in dieser Bibliothek getätigten Migrationsaufwands keinen Beitrag für die Integration des pta_tool, wohl aber für die Portierung weiterer Werkzeuge, begünstigt. Eine ähnliche Aufwand-Nutzen-Abschätzung wird folglich auch bei der Migration der libiml vermutet.

Nachdem im März 2018 (5/7 der Projektzeit) immer noch kein compilierbarer Stand des pta_tools vorhanden war, wurde beschlossen, die beiden Abhängigkeiten, das andersen– und das das-Tool, aus dem Buildprozess zu entfernen. Hierzu wurden die entsprechenden Quellcodedateien in einen *src-unused* Ordner verschoben. Selbiges Vorgehen wurde auch schon in der Vorgängerarbeit [Przytarski, 2016] durchgeführt. Mit dieser Entscheidung konnte der Fortschritt der Migration spürbar beschleunigt werden.

Weiterhin ist zu berücksichtigen, dass die grünen Komponenten, insbesondere die libiml, zwar compilierbar sind, jedoch nicht alle ursprünglichen Quelldateien beinhalten. Ähnlich wie dies für das andersen– und das-Tool bereits durchgeführt worden ist, sind alle nicht compilierbaren Einheiten in einen *src-unused*-Ordner verschoben und somit vom Buildprozess ausgeklammert worden. Folglich fehlten der libiml auch viele für das pta_tool notwendige Dateien, weshalb sie aus der Sicht des pta_tools eher ein roter Baustein sind. Tabelle 8 listet alle Quelldateien auf, mit welchen die libiml erweitert wurde. Die zweite Spalte der Tabelle beziffert die Anzahl der Codezeilen (LoC) der neuen Dateien.

Neben den beschriebenen Dateien, die explizit aus dem Buildprozess ausgeschlossen waren, beinhalteten die compilierbaren Pakete sehr viele Funktionen und Prozeduren, deren *Bodys* vollständig auskommentiert sind (ähnlich wie in Quellcode 22 dargestellt). Somit lassen sich Dateien compilieren, die in Abbildung 7.1 Teil einer grünen Komponente sind, obwohl diese keinen validen Code beinhalten. Spätestens zur Laufzeit wird

Dateien	LoC
src-unused/df_mapping_facts.adb	743
src-unused/df_mapping_facts.ads	283
src-unused/iml_reflection-dynamic.adb	576
src-unused/iml_reflection-dynamic.ads	119
utilities-unused/callgraph_scc_maps.adb	67
utilities-unused/callgraph_scc_maps.ads	46
utilities-unused/iml-cg_analysis.adb	977
utilities-unused/iml-cg_analysis.ads	353
utilities-unused/iml-dispatchers.adb	111
utilities-unused/iml-dispatchers.ads	119
utilities-unused/iml-interfaces-pattern_cg.adb	291
utilities-unused/iml-interfaces-pattern_cg.ads	150
utilities-unused/iml_nonlocals.adb	456
utilities-unused/iml_nonlocals.ads	172
utilities-unused/iml-pattern_cg_analysis.adb	1453
utilities-unused/iml-pattern_cg_analysis.ads	537
Summe:	6453

Tabelle 8: Hinzugenommen Quellcodedateien in die Bibliothek libiml

man hier bei den betroffenen Werkzeugen auf Probleme stoßen.

Das pta_tool benötigt nun viele dieser Codepassagen. Generell ist es schwierig, genau zu beziffern, wie viele ehemals auskommentierte Codezeilen im Kontext dieser Arbeit zusätzlich in den Buildprozess aufgenommen worden sind. Einen ungefähren Überblick ermöglicht aber die folgende Messung. In der Regel wurden die auskommentierten *Bodys* mit einer „*todo*“-Ausnahme abgeschlossen. Sucht man nun nach git Änderungen, bei denen genau diese Ausnahme entfernt wird, gelangt man auf 178 Codezeilen. Zu beachten ist hierbei, dass es sich nicht nur um 178 Zeilen, sondern um 178 Unterprogramme, die zusätzlich in den Bauhaus-Build aufgenommen wurden, handelt. Der verwendete Suchaufruf ist in Quellcode 39 dargestellt.

```
1 git diff master..HEAD | grep  -e '\(- *raise Program_Error with "todo
    ";\)\|\(\.adb\)' | grep -c 'todo'
2 178
```

Quellcode 39: Messung der entfernten Zeilen mit der todo Ausnahme

Lässt man den letzten grep Aufruf mit der Zähloption *-c* weg und betrachtet die betroffenen Dateien, so ist festzustellen, dass es keine Schnittmenge zu den in Tabelle 8 gelisteten Dateien gibt.

Außerdem verursacht die enge Verzahnung des pta_tools im Bauhaus-Projekt, dargestellt in Abbildung 7.1, auch erhebliche Probleme bei den notwendigen Migrationsschritten. Die dargestellten Abhängigkeiten bedeuten Unterprogrammaufrufe zwischen

den Paketen. Parameter und Rückgabewerte werden ausgetauscht. Die in Kapitel 4.1 beschriebenen Folgefehler können sich somit auf bis dahin funktionsfähige Dateien auswirken. Eine reparierte Codezeile hatte oftmals mehrere hundert neue Compilerfehler zur Folge. Diese Folgefehler erklären auch die große Differenz der insgesamt bearbeiteten Dateien (177) und der neu hinzugenommen, die in diesem Kapitel beschrieben sind.

7.1.2. Bestehende Fehler im Projekt

Im Rahmen der Integration wurden ebenfalls Fehler im Projekt entdeckt, die bei vorherigen Migrationsarbeiten entstanden sind. Da die betroffenen Komponenten erstmals mit dieser Masterarbeit verwendet werden konnten, sind diese Fehler bisher unentdeckt geblieben. Hierzu zählen die beschriebenen Probleme in Kapitel 4.10.1, Kapitel 4.10.3 oder Kapitel 4.11.1.

7.2. Erreichte Ziele

In Kapitel 3.2 sind die ursprünglich geplanten Ziele dieser Masterarbeit beschrieben. Es wurde früh deutlich, dass die Migration von vier Werkzeugen innerhalb der angesetzten Projektzeit nicht umsetzbar ist. Kapitel 7.1 erläutert die Hintergründe für diese Einschätzung. Daher konzentriert sich diese Arbeit auf die in Kapitel 3.2.1 aufgezeigten Ziele. Die erfolgreiche Integration eines Werkzeuges wurde am Beispiel des pta_tools gezeigt. Ebenso wurde die Bibliothek lib_concurrency vollständig migriert sowie die libiml erweitert. Ein Großteil dieses Aufwandes lässt sich mit dem funktionalen Ergebnis dieser Arbeit, dem pta_tool, nicht messen. Es wird vermutet, dass die Migration weiterer Werkzeuge mithilfe der in diesem Projekt geleisteten Vorarbeit deutlich schneller erfolgen kann.

Eine entsprechende Analyse gegen Ende dieser Arbeit bestätigt diese Sichtweise. Die Werkzeuge thread_tool und raceq besitzen identische Abhängigkeiten wie das pta_tool. Beide Werkzeuge bestehen lediglich aus einer einzelnen Quelltextdatei. Der Migrationsaufwand ist nun vermutlich deutlich geringer. Ohne das vorgeschaltete Werkzeug iml2cfg, welches eine weitere Bibliothek benötigt, ist diese Integration jedoch nicht zielführend. Analog zu thread_tool und raceq besitzen auch die Werkzeuge in der folgenden Liste identische Abhängigkeiten wie das pta_tool und profitieren bei einer zukünftigen Migration von dieser Masterarbeit.

- *projects/tools/stats*
- *projects/tools/kcg_stats*
- *projects/tools/dra_filter*
- *projects/tools/dump_paths*
- *projects/tools/ssa2cg*

- *projects/tools/sat_constraints*

Neben dem pta_tool und der lib_concurrency entstand mit dieser Arbeit auch eine Dokumentation für weitere Migrationsprojekte. Es wurden Konvertierungsregeln erarbeitet, die eine Portierung von IML-Programmkonstrukten in die *SKilL*-Version des Bauhaus-Projektes ermöglichen. Die strukturierte Auflistung der Migrationsthemen und möglicher Fehlerbilder werden ebenfalls einen wesentlichen Beitrag zur Beschleunigung zukünftiger Migrationsprojekte darstellen.

Entsprechend der Anforderungen wurde das pta_tool mit einer automatischen Testsuite mithilfe ausgewählter Testdateien getestet. Weiterhin ist das implementierte Testwerkzeug nicht auf die Data Race Werkzeuge beschränkt, sondern kann für die automatische Ausführung jeglicher Bauhauswerkzeuge verwendet werden.

7.3. Offene Punkte

Um eine Data Race Analyse für C-Programme durchzuführen, müssen die verbleibenden drei Werkzeuge iml2cfg, thread_tool und raceq auf gleiche Weise migriert werden. Wie im vorherigen Kapitel 7.2 beschrieben, dürfte die Integration der beiden Letzgenannten vergleichsweise schnell erfolgen. Iml2cfg benötigt jedoch noch die Bibliothek libdataflow, die ebenfalls migriert werden muss.

In Kapitel 6.5 sind zwei Testfälle beschrieben, die in beiden pta_tool Versionen unterschiedliches Verhalten aufzeigen. Die IML-Version endet in einer Endlosschleife, die *SKilL*-Version erzeugt eine scheinbar valide Binärdatei. Die Hintergründe für diese Abweichung müssen noch analysiert werden.

Weiterhin wurden im Kontext dieser Arbeit immer wieder Annahmen getroffen, um die Fertigstellung eines ersten migrierten Standes im Rahmen der Arbeit zu ermöglichen. Sobald mit der Ausgabe von raceq ein lesbares und leicht zu vergleichendes Ergebnis vorliegt, müssen diese Schritte nochmals validiert werden.

Die folgende Aufzählung beinhaltet die relevanten Aspekte. Die ersten drei Punkte beinhalten dabei Validierungsmaßnahmen, welche bei Abschluss dieser Arbeit noch nicht durchgeführt wurden. Sollten entsprechende Tests erfolgreich verlaufen, besteht hier keinerlei Handlungsbedarf.

- Es ist noch unklar ob der Austausch eines AVL-Baumes mit dem Ada-*Hashed_Sets* eine Auswirkung auf das Programmverhalten hervorruft (siehe Kapitel 4.7.3). Im Falle von Testabweichungen sollte dieser Migrationsschritt validiert werden.
- In Kapitel 4.7.2 wurde der Hintergrund der Datenstruktur *Resizable_Array* und die Notwendigkeit eines gültigen und ungültigen Array-Bereichs nicht vollkommen verstanden. Im Zuge dessen besteht eine Unsicherheit bei der Konvertierung der Funktion *Set_Range*.

- Ebenfalls in Kapitel 4.7.2 wurde bei einer Variable die Set-Eigenschaft ignoriert. Dies betrifft die Konvertierung des Unterprogramms *Insert_Single*. Sollte diese Vereinfachung negative Auswirkung auf die Analyse besitzen, muss dieser Migrationsschritt angepasst werden.

- Um die Portierung des pta_tools im gegebenen Zeitrahmen abzuschließen, wurde eine vereinfachte Generierung der Less-Implementierung angewandt. Langfristig sollte hier ein besserer Ansatz verfolgt werden (Siehe Kapitel 4.14.2).

- Kapitel 5.5 beschreibt, dass die Assert Prüfungen in einer .c-Datei auskommentiert wurden. Hier kommt es offensichtlich zu einer Abweichung, welche noch analysiert werden muss. Führt man die durchgeführten Tests aus Kapitel 6.4 ohne das Macro NDEBUG aus, so scheitern ca. 60 % der Tests an der selben Assert-Überprüfung.

- Die durchgefallenen Testfälle aus Kapitel 6 sind noch nicht vollständig analysiert.

7.4. Ausblick

Mit der gelungenen Integration der lib_concurrency und des pta_tools wurde der Austausch der Zwischendarstellung innerhalb des Bauhaus-Projektes erfolgreich fortgeführt. Kapitel 7.2 listet eine Reihe von Werkzeugen auf, die aufgrund der erzielten Ergebnisse in anschließenden Projekten effizient portiert werden können. Weiterhin können die gewonnenen Erkenntnisse dieser Arbeit gewinnbringend in weiteren Migrationsprojekten genutzt werden.

Anhang

A. Liste von verschobenen Dateien

Im Rahmen dieser Masterarbeit wurden Dateien verschoben und auf diese Weise vom Buildprozess ausgeschlossen. Dieses Kapitel beinhaltet eine Übersicht der verschobenen Dateien.

Verschobene Andersen Dateien

Die im Folgenden gelisteten Dateien sind aus dem pta_tool-Buildprozess ausgeklammert, indem sie von dem Ordner *projects/libs/andersen/src* nach *projects/libs/andersen/src_unused* verschoben sind:

```
andersen_analysis.adb, andersen_analysis.ads, andersen_calls.adb,
andersen_calls.ads, andersen_data.adb, andersen_data.ads,
andersen_ecr_field_mappings.ads, andersen_ecr_method_mappings.ads,
andersen_ecr_type_mappings.ads, andersen_ecrs.adb, andersen_ecrs.ads,
andersen_element.adb, andersen_element.ads,
andersen_element_mappings.ads, andersen_expression.adb,
andersen_expression.ads, andersen_generic_mappings.adb,
andersen_generic_mappings.ads, andersen_interface.adb,
andersen_interface.ads, andersen_mappings.adb, andersen_mappings.ads,
andersen_results.adb, andersen_results.ads,
andersen_unknown_call_mappings.ads,
andersen_unknown_element_mappings.ads, andersen_utils.adb,
andersen_utils.ads, andersen_variable_mappings.ads,
andersen_virtual_call_mappings.ads
```

Verschobene Das Dateien

Die im Folgenden gelisteten Dateien sind aus dem pta_tool-Buildprozess ausgeklammert, indem sie von dem Ordner *projects/libs/das/src* nach *projects/libs/das/src_unused* verschoben sind:

```
das_analysis.adb, das_analysis.ads, das_data.adb, das_data.ads,
das_expression.adb, das_expression.ads, das_interface.adb,
das_interface.ads, das_routine.adb, das_routine.ads,
das_structure.adb, das_structure.ads
```

Verschobene PSI Dateien

Die im Folgenden gelisteten Dateien sind aus dem pta_tool-Buildprozess ausgeklammert, indem sie von dem Ordner *projects/libs/concurrency/src* nach *projects/libs/concurrency/src_unused* verschoben sind:

psi_handling.adb, psi_handling.adb, psix_placement.adb,
psi_placement.ads

B. Auflistung der Testdateien

In diesem Kapitel werden die Dateien der in Kapitel 6.4 beschriebenen Testsets aufgelistet. Aus Gründen der Übersicht sind lediglich die genutzten .iml.sf-Dateien gelistet. Die IML-Version des pta_tools nutzt die entsprechenden .iml-Dateien.

Testset 1: Data Race Beispiele

functionPointer_FalsePositive_Keul.iml.sf, helloWorld.iml.sf,
pThreadTwiceSquare.iml.sf, pThreadTwiceSquare_Mutex.iml.sf,

Testset 2: Existierende Open Source C-Dateien

Waddress-1.iml.sf, Waddress-2.iml.sf, Warray-bounds-2.iml.sf,
Warray-bounds-5.iml.sf, Wattributes-2.iml.sf, Wattributes.iml.sf,
Wbool-compare-1.iml.sf, Wbool-compare-2.iml.sf, Wbool-compare-3.iml.sf,
Wbool-operation-1.iml.sf, Wbuiltin-declaration-mismatch-1.iml.sf,
Wcast-align.iml.sf, Wcast-function-type.iml.sf, Wcast-qual-1.iml.sf,
Wconversion-real.iml.sf, Wdangling-else-1.iml.sf,
Wdangling-else-2.iml.sf, Wdangling-else-3.iml.sf,
Wdangling-else-4.iml.sf, Wduplicated-branches-1.iml.sf,
Wduplicated-branches-10.iml.sf, Wduplicated-branches-11.iml.sf,
Wduplicated-branches-12.iml.sf, Wduplicated-branches-13.iml.sf,
Wduplicated-branches-14.iml.sf, Wduplicated-branches-3.iml.sf,
Wduplicated-branches-4.iml.sf, Wduplicated-branches-5.iml.sf,
Wduplicated-branches-6.iml.sf, Wduplicated-branches-7.iml.sf,
Wduplicated-branches-8.iml.sf, Wduplicated-branches-9.iml.sf,
Wduplicated-cond-1.iml.sf, Wduplicated-cond-2.iml.sf,
Wduplicated-cond-3.iml.sf, Wduplicated-cond-4.iml.sf,
Wfloat-conversion.iml.sf, Wformat-pr84258.iml.sf,
Wimplicit-fallthrough-1.iml.sf, Wimplicit-fallthrough-11.iml.sf,
Wimplicit-fallthrough-12.iml.sf, Wimplicit-fallthrough-13.iml.sf,
Wimplicit-fallthrough-14.iml.sf, Wimplicit-fallthrough-15.iml.sf,
Wimplicit-fallthrough-16.iml.sf, Wimplicit-fallthrough-17.iml.sf,
Wimplicit-fallthrough-18.iml.sf, Wimplicit-fallthrough-19.iml.sf,
Wimplicit-fallthrough-2.iml.sf, Wimplicit-fallthrough-21.iml.sf,
Wimplicit-fallthrough-22.iml.sf, Wimplicit-fallthrough-23.iml.sf,
Wimplicit-fallthrough-24.iml.sf, Wimplicit-fallthrough-25.iml.sf,
Wimplicit-fallthrough-26.iml.sf, Wimplicit-fallthrough-27.iml.sf,
Wimplicit-fallthrough-28.iml.sf, Wimplicit-fallthrough-29.iml.sf,
Wimplicit-fallthrough-3.iml.sf, Wimplicit-fallthrough-30.iml.sf,
Wimplicit-fallthrough-31.iml.sf, Wimplicit-fallthrough-32.iml.sf,

Wimplicit-fallthrough-33.iml.sf, Wimplicit-fallthrough-34.iml.sf,
Wimplicit-fallthrough-35.iml.sf, Wimplicit-fallthrough-36.iml.sf,
Wimplicit-fallthrough-4.iml.sf, Wimplicit-fallthrough-5.iml.sf,
Wimplicit-fallthrough-8.iml.sf, Wimplicit-fallthrough-9.iml.sf,
Wint-in-bool-context-2.iml.sf, Wint-in-bool-context-3.iml.sf,
Wint-in-bool-context.iml.sf, Wint-to-pointer-cast-1.iml.sf,
Wint-to-pointer-cast-2.iml.sf, Wint-to-pointer-cast-3.iml.sf,
Wlogical-not-parentheses-1.iml.sf, Wlogical-not-parentheses-2.iml.sf,
Wlogical-not-parentheses-3.iml.sf, Wlogical-op-1.iml.sf,
Wlogical-op-2.iml.sf, Wlogical-op-3.iml.sf,
Wmemset-transposed-args1.iml.sf, Wmisleading-indentation-2.iml.sf,
Wmisleading-indentation-3.iml.sf, Wmisleading-indentation.iml.sf,
Wmultistatement-macros-1.iml.sf, Wmultistatement-macros-10.iml.sf,
Wmultistatement-macros-11.iml.sf, Wmultistatement-macros-12.iml.sf,
Wmultistatement-macros-13.iml.sf, Wmultistatement-macros-2.iml.sf,
Wmultistatement-macros-3.iml.sf, Wmultistatement-macros-4.iml.sf,
Wmultistatement-macros-5.iml.sf, Wmultistatement-macros-6.iml.sf,
Wmultistatement-macros-7.iml.sf, Wmultistatement-macros-9.iml.sf,
Wno-builtin-declaration-mismatch-1.iml.sf, Wparentheses-1.iml.sf,
Wpointer-arith-1.iml.sf, Wpointer-compare-1.iml.sf,
Wrestrict-2.iml.sf, Wrestrict-3.iml.sf,
Wsequence-point-1.iml.sf, Wshift-count-negative-1.iml.sf,
Wshift-count-negative-2.iml.sf, Wshift-count-overflow-1.iml.sf,
Wshift-count-overflow-2.iml.sf, Wshift-negative-value-1.iml.sf,
Wshift-negative-value-2.iml.sf, Wshift-negative-value-3.iml.sf,
Wshift-negative-value-4.iml.sf, Wshift-negative-value-5.iml.sf,
Wshift-negative-value-6.iml.sf, Wshift-overflow-1.iml.sf,
Wshift-overflow-2.iml.sf, Wshift-overflow-3.iml.sf,
Wshift-overflow-4.iml.sf, Wshift-overflow-5.iml.sf,
Wsign-compare-1.iml.sf, Wsizeof-pointer-div.iml.sf,
Wsizeof-pointer-memaccess1.iml.sf, Wsizeof-pointer-memaccess2.iml.sf,
Wsizeof-pointer-memaccess3.iml.sf, Wstringop-overflow.iml.sf,
Wstringop-truncation-3.iml.sf, Wstringop-truncation-4.iml.sf,
Wstringop-truncation.iml.sf, Wswitch-unreachable-2.iml.sf,
Wswitch-unreachable-3.iml.sf, Wswitch-unreachable-4.iml.sf,
Wtautological-compare-1.iml.sf, Wtautological-compare-2.iml.sf,
Wtautological-compare-3.iml.sf, Wtautological-compare-4.iml.sf,
Wtautological-compare-5.iml.sf, Wtautological-compare-6.iml.sf,
Wtautological-compare-7.iml.sf, Wunused-function-1.iml.sf,
Wunused-local-typedefs.iml.sf, Wunused-var-1.iml.sf,
Wunused-var-10.iml.sf, Wunused-var-11.iml.sf, Wunused-var-12.iml.sf,
Wunused-var-13.iml.sf, Wunused-var-14.iml.sf, Wunused-var-2.iml.sf,
Wunused-var-4.iml.sf, Wunused-var-5.iml.sf, Wunused-var-6.iml.sf,
Wunused-var-7.iml.sf, Wunused-var-8.iml.sf, Wunused-var-9.iml.sf,
addrtmp.iml.sf, attr-aligned-1.iml.sf, attr-may-alias-1.iml.sf,
attr-may-alias-2.iml.sf, attr-nocf-check-1.iml.sf,

attr-nocf-check-2.iml.sf, attr-nocf-check-3.iml.sf,
attr-nonstring-1.iml.sf, attr-nonstring-2.iml.sf,
attr-nonstring-3.iml.sf, attr-nonstring-4.iml.sf,
attr-nonstring-5.iml.sf, attr-opt-1.iml.sf,
attr-simd-2.iml.sf, attr-simd-4.iml.sf, attr-simd-5.iml.sf,
attr-simd.iml.sf, attr-used-2.iml.sf, attr-used.iml.sf,
attributes-2.iml.sf, attributes-3.iml.sf,
builtin-arith-overflow-2.iml.sf, conflict-markers-2.iml.sf,
conflict-markers-7.iml.sf, conflict-markers-8.iml.sf,
conflict-markers-9.iml.sf, convert-vec-1.iml.sf,
cxxbitfields-3.iml.sf, cxxbitfields-6.iml.sf,
dump-ada-spec-1.iml.sf, dump-ada-spec-10.iml.sf,
dump-ada-spec-11.iml.sf, dump-ada-spec-12.iml.sf,
dump-ada-spec-13.iml.sf, dump-ada-spec-2.iml.sf,
dump-ada-spec-3.iml.sf, dump-ada-spec-4.iml.sf,
dump-ada-spec-5.iml.sf, dump-ada-spec-6.iml.sf,
dump-ada-spec-7.iml.sf, dump-ada-spec-8.iml.sf,
dump-ada-spec-9.iml.sf, fcf-protection-1.iml.sf,
fcf-protection-2.iml.sf, fcf-protection-3.iml.sf,
fcf-protection-4.iml.sf, fcf-protection-5.iml.sf,
fcf-protection-6.iml.sf, fcf-protection-7.iml.sf,
ffile-prefix-map.iml.sf, fmacro-prefix-map.iml.sf,
fold-bitand-4.iml.sf, fold-divmul-1.iml.sf,
fold-masked-cmp-1.iml.sf, fold-masked-cmp-2.iml.sf,
fold-masked-cmp-3.iml.sf, init-vec-1.iml.sf, memset-array.iml.sf,
missing-header-3.iml.sf, missing-header-4.iml.sf,
nonnull-1.iml.sf, nonnull-2.iml.sf, nonnull-3.iml.sf,
opaque-vector.iml.sf, patchable_function_entry-decl.iml.sf,
patchable_function_entry-default.iml.sf,
patchable_function_entry-definition.iml.sf,
pr19807-1.iml.sf, pr20318.iml.sf, pr27336.iml.sf, pr28656.iml.sf,
pr30020.iml.sf, pr33763.iml.sf, pr35503-1.iml.sf, pr35503-2.iml.sf,
pr35503-3.iml.sf, pr36282-1.iml.sf, pr36282-2.iml.sf,
pr36282-3.iml.sf, pr36282-4.iml.sf, pr36513-2.iml.sf, pr36513.iml.sf,
pr37743.iml.sf, pr41779.iml.sf, pr41935.iml.sf, pr42674.iml.sf,
pr43395.iml.sf, pr43690.iml.sf, pr43772.iml.sf, pr43942.iml.sf,
pr44832.iml.sf, pr46562-2.iml.sf, pr46562.iml.sf, pr48418.iml.sf,
pr49706-2.iml.sf, pr49706.iml.sf, pr50459.iml.sf, pr51294.iml.sf,
pr51712.iml.sf, pr52177.iml.sf, pr52181.iml.sf, pr53037-5.iml.sf,
pr53633.iml.sf, pr53874.iml.sf, pr54486.iml.sf, pr54988.iml.sf,
pr55619.iml.sf, pr55771.iml.sf, pr56302.iml.sf, pr56493.iml.sf,
pr56566.iml.sf, pr56607.iml.sf, pr57371-1.iml.sf, pr57371-2.iml.sf,
pr57371-4.iml.sf, pr57653-2.iml.sf, pr58346-1.iml.sf, pr58346-2.iml.sf,
pr58346-3.iml.sf, pr59032.iml.sf, pr59223.iml.sf, pr60101.iml.sf,
pr60156.iml.sf, pr60226.iml.sf, pr60689.iml.sf, pr61405.iml.sf,
pr61534-1.iml.sf, pr61553.iml.sf, pr62199-2.iml.sf, pr62199.iml.sf,

pr65040.iml.sf, pr65120.iml.sf, pr65556.iml.sf, pr65830.iml.sf,
pr66208.iml.sf, pr67639.iml.sf, pr67653.iml.sf, pr68582.iml.sf,
pr68657-1.iml.sf, pr68657-2.iml.sf, pr68657-3.iml.sf, pr68833-1.iml.sf,
pr68833-2.iml.sf, pr69126-2-long.iml.sf, pr69126-2-short.iml.sf,
pr69126.iml.sf, pr69543-1.iml.sf, pr69543-2.iml.sf, pr69543-3.iml.sf,
pr69543-4.iml.sf, pr69558-1.iml.sf, pr69558-2.iml.sf, pr69558-3.iml.sf,
pr69558-4.iml.sf, pr69558.iml.sf, pr69669.iml.sf, pr69733.iml.sf,
pr69764.iml.sf, pr69797.iml.sf, pr70144-1.iml.sf, pr70144-2.iml.sf,
pr70297.iml.sf, pr70336.iml.sf, pr70756-2.iml.sf, pr71372.iml.sf,
pr71654.iml.sf, pr77624-1.iml.sf, pr77624-2.iml.sf, pr79428-3.iml.sf,
pr79641.iml.sf, pr81052.iml.sf, pr82112.iml.sf, pr83059.iml.sf,
pr84293.iml.sf, pr84982.iml.sf, pr85156.iml.sf, restrict-1.iml.sf,
restrict-2.iml.sf, restrict-4.iml.sf, rotate-1.iml.sf, rotate-1a.iml.sf,
rotate-2.iml.sf, rotate-2a.iml.sf, rotate-3.iml.sf, rotate-3a.iml.sf,
rotate-4.iml.sf, rotate-4a.iml.sf, rotate-6.iml.sf, rotate-6a.iml.sf,
rotate-7.iml.sf, rotate-7a.iml.sf, rotate-8.iml.sf, scal-to-vec2.iml.sf,
taskloop-1.iml.sf, transparent-union-1.iml.sf, uninit-17.iml.sf,
uninit-D-O0.iml.sf, uninit-D.iml.sf, uninit-E-O0.iml.sf, uninit-E.iml.sf,
uninit-F-O0.iml.sf, uninit-F.iml.sf, uninit-G-O0.iml.sf, uninit-G.iml.sf,
uninit-pr51010.iml.sf, vector-3.iml.sf, vector-4.iml.sf,
vector-compare-1.iml.sf, vector-compare-2.iml.sf, vector-compare-3.iml.sf,
vector-init-1.iml.sf, vector-init-2.iml.sf, vector-scalar-2.iml.sf,
vector-scalar.iml.sf, vla-1.iml.sf, warn-ommitted-condop.iml.sf

Testset 3: Testdateien der Vorgängerarbeit

binary.iml.sf, bzip2.iml.sf, clonetypes.iml.sf, dowhile.iml.sf,
duplicates1.iml.sf, duplicates2.iml.sf, duplicates3.iml.sf,
easyzlib.iml.sf, empty.iml.sf, fibonacci.iml.sf, floats.iml.sf,
func.iml.sf, gzip.iml.sf, hello.iml.sf, ifs.iml.sf,
levenshtein.iml.sf, matrices.iml.sf, mpc.iml.sf, names.iml.sf,
nesting.iml.sf, nesting2.iml.sf, oggenc.iml.sf, operators.iml.sf,
overlapping.iml.sf, parg.iml.sf, ph7.iml.sf, sequence.iml.sf,
sqlite3.iml.sf, stmr.iml.sf, switch.iml.sf, test-mccabe.iml.sf,
tweetnacl.iml.sf, unary.iml.sf, unqlite.iml.sf, ylog.iml.sf,
zlib_amalg.iml.sf

C. Beispielskript zur automatischen Fehlerbehebung

Einige Compilerfehler lassen sich effizient mithilfe von Skripten automatisch beheben. Im Folgenden ist ein awk-Skript zum Auflösen der Ada Präfix-Notation angegeben.

Wrapper-Skript

Es wird empfohlen, die awk-Skripte nur auf den betroffenen Quellcodedateien auszuführen. Auf diese Weise bleiben funktionierende Ada-Dateien von den Modifikationen unberührt. Die betroffenen Quellcodedateien können zum Beispiel aus der Konsolenausgabe des Bauhausbuilds geparst werden. Das Skript in Quellcode 40 erwartet eine Datei mit Pfaden zu den Quellcodedateien, die durch das angegebene awk-Skript in Zeile 12 modifiziert werden sollen.

```
1  # Example call:
2  #./runawk.awk Files.txt
3  # Files.txt contains path to all files which should be manipulated
4  filename="$1"
5  while read -r line
6  do
7      echo "editing $line \n"
8      if test "$line" = awkscr; then
9          echo "not editing awkscript!"
10     else
11         # Replace this line to call another awk script:
12         awk -f resolvePrefix.awk $line > tmp.txt
13     cp tmp.txt $line
14     fi
15 done < "$filename"
```

Quellcode 40: Beispielhaftes Bash-Skript zum Ausführen der awk-Skripte. Das Skript erwartet als Argument eine Textdatei mit Pfaden der zu den Sourcedateien.

Auflösen der Präfix-Notation

Mit dem abgebildeten Skript können Funktionsaufrufe, die die Ada Präfix-Notation verwenden, in explizite Funktionsaufrufe umgewandelt werden. Beispielhafte Skriptaufrufe können den Kommentarzeilen entnommen werden.

```
1  BEGIN {
2      PACKAGES[0] = "PTA_Objects"
3      PACKAGES[1] = "PTA_DF_Patterns"
4      PACKAGES[2] = "Thread_DF_Patterns"
5  }
6  # Example 1: (cast with . notation)
7  # replacePrefixFast("Dummy_First_Refparam_Field","PTA_Objects",".Dynamic_PTA_Object_Map")
8  # Same Call without "Fast". In that case, you must manually specify the
9  # corresponding variable
10 # replacePrefix("Map","Dummy_First_Refparam_Field","PTA_Objects",".Dynamic_PTA_Object_Map
       ")
11
12 # Example 2: (cast with function call)
13 # use cast with function call:
14 # replacePrefixFast("Function_Of","PTA_Objects","Siml.To_PTA_Object_Map (Skill.Types.
       Annotation")
15
16 # Example 3: (no cast needed)
17 # replacePrefixFast("Function_Of","PTA_Objects","")
18
19 # Variable.Funcname(...)
20 # will be converted into something like:
21 # Package.Funcname(Variable, ...)
22 # if needed a cast is used to convert variable
23 function replacePrefix(VARIABLE,FUNCNAME,PACKAGE,CAST) {
24
```

```
25      regExprPara = VARIABLE"."FUNCNAME" \\("
26      regExprSing = VARIABLE"."FUNCNAME"[\\);, ]"
27
28      if (match($0,"[ ()]"regExprPara)){
29          # parameters follow in the same line.
30          regExpr = regExprPara;
31          lastSymbol = ", ";
32      }
33      else if (match($0,"[ ()]"regExprSing)){
34          # there exit one parameter.
35          lastSymbol = ")" substr($0,RSTART+RLENGTH-1,1);
36          regExpr = regExprSing;
37      }
38      else {
39          print
40          next
41      }
42
43      if (CAST == "") {
44      # should work without cast
45          sub(regExpr,PACKAGE"."FUNCNAME" ("VARIABLE lastSymbol,$0)
46      }
47      else if (CAST ~ "^\\..*"){
48          # cast with . notation
49          sub(regExpr,PACKAGE"."FUNCNAME" ("VARIABLE CAST lastSymbol,$0)
50      }
51      else {
52          # normal cast with function call:
53          sub(regExpr,PACKAGE"."FUNCNAME" ("CAST" ("VARIABLE"))"lastSymbol,$0)
54      }
55      print $0
56  }
57
58  # same as replacePrefix() but the function will automatically detect
59  # the variable name. after that replacePrefix is called.
60  function replacePrefixFast(FUNCNAME,PACKAGE,CAST) {
61  #print "[)]([a-zA-Z_.])*."FUNCNAME" \\("
62
63      # parameter follow in the same line
64      if(hit = match($0,"([^)]|[ -])([a-zA-Z_.])*\\."FUNCNAME" \\(")){
65          # +1 to skip leading whitespace
66          wholePattern = substr($0,RSTART+1,RLENGTH)
67          # grep the beginning until function:
68          match(wholePattern,"."FUNCNAME" \\(")
69      }
70      else if (hit = match($0,"([^)]|[ -])([a-zA-Z_.])*\\."FUNCNAME"[\\);, ]")){
71      # there exist only one parameter. will probably terminate with ; or )
72          # +1 to skip leading whitespace
73          wholePattern = substr($0,RSTART+1,RLENGTH)
74          # grep the beginning until function:
75          match(wholePattern,"."FUNCNAME"[\\);, ]")
76      }
77      if (hit > 0) {
78      # found prefix pattern to resolve:
79          variable = substr(wholePattern,0,RSTART-1);
80          #print PACKAGE
81          if (variable == PACKAGES[0] ||
82              variable == PACKAGES[1] ||
83              variable == PACKAGES[2]) {
84              # may be the case that the prefix notation is already fixed.
85              print
86          } else {
87              replacePrefix(variable,FUNCNAME,PACKAGE,CAST)
88          }
89      }
90      else { print }
91  }
```

Quellcode 41: Skript *resolvePrefix.awk* zum automatischen Umwandeln der Präfix-Notation in explizite Funktionsaufrufe. Falls gewünscht, wird noch ein Cast hinzugefügt.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Glossary

Ada 2005 ISO Standard der Programmiersprache Ada. Nachfolger von Ada 95. XI, 16

Ada 2012 ISO Standard der Pogrammiersprache Ada [ISO/IEC, 2012]. Nachfolger von Ada 2005. 29, 40

andersen Werkzeug für die Zeigeranalyse nach Andersen. 42, 59

awk Unix Skriptsprache benannt nach den drei Autoren Alfred V. **A**ho, Peter J. **W**einberger und Brian W. **K**ernighan. VI

cafe++ Bauhaus Compiler für die statische Code-Analyse. cafe++ erzeugt aus einer .c-Datei eine .iml-Datei. XI, 8

cafecc Bauhaus Werkzeug welches unter anderem den Compiler cafe++ aufruft.. XI, 8, 53, 58

codegen Codegenerator, der eine *SKilL*-Spezifikation einliest und darauf die API für den IML-Graphen generiert [Przytarski, 2016, Kapitel 5]. X, XI, 11, 12, 37, 44, 45, 51

das Werkzeug für die Zeigeranalyse nach Das. 42, 59

ecr Werkzeug für die Zeigeranalyse nach Steensgard. 42

git Verteiltes Versionsverwaltungssystem, mit dem auch das Bauhaus-Projekt verwaltet wird. 60

grep Programm zum Suchen definierter Zeichenketten. Die Abkürzung steht für **g**lobal **r**egular **e**xpression **p**rint. 60

iml2cfg Werkzeug für die Data Race Analyse. Wird nach pta_tool und vor thread_tool aufgerufen. XII, XIII, 12, 61, 62

iml2sf Werkzeug, das aus einem IML-Graphen (.iml-Datei) einen *SKilL*-Graphen .iml.sf-Datei generiert. XII

.iml-Datei IML-Binärdatei, die den IML-Graphen beinhaltet. Wird mithilfe des Werkzeuges cafecc generiert. II, XI, XII, 53

imlgen4 Werkzeug bestehend aus den Werkzeugen specgen und codegen. 12

imlink Bauhaus Linker, der mit cafe++ generierte *.iml-Dateien bindet. 8

imlmetrics Bauhaus Werkzeug zum Erfassen von Codemetriken. 52, 54, 56, 57

.iml.sf-Datei *SKilL*-Binärdatei, die mithilfe von iml2sf aus einer .iml-Datei generiert wird. II, XI, XII, 53, 55–58

imlstat Analysewerkzeug der Bauhaustoolsuite. 52

IML-Version Ursprüngliche Bauhaus-Version, welche ausschließlich IML als Zwischensprache verwendet. Sie dient als Referenz für die *SKilL*-Migration dieser Masterarbeit. IX, XII, 11, 14, 19, 20, 31, 37, 38, 44, 53, 56

lib_clonedetection Bauhaus Bibliothek, die unter anderem für das pta_tool benötigt wird. 58

lib_concurrency Bauhaus Bibliothek, die unter anderem für das pta_tool benötigt wird. 58, 59, 61–63

libdataflow Bauhaus Bibliothek, die unter anderem für das Werkzeug iml2cfg benötigt wird. 62

libiml Bauhaus Bibliothek, besteht unter anderem aus der IML-und *SKilL*-Spezifikationen sowie den daraus generierten Dateien. VIII, 37, 43, 58–61

lib_ptaquery Bauhaus Bibliothek, die unter anderem für das pta_tool benötigt wird. 42

lib_reuse Bauhaus Bibliothek, die von zahlreichen Werkzeugen verwendet wird. 58

MISRA MISRA ist ein C-Programmierstandard aus der Automobilindustrie. Die Abkürzung steht für **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation. 8

.pta-Datei Binäre Ausgabedatei des pta_tool's. VIII, XII, 56, 57

pta_tool Werkzeug für die Data Race Analyse. Liest eine .iml-Datei bzw. .iml.sf-Datei ein und erzeugt eine .pta-Datei Datei. I, II, VIII, XI, XII, 12, 19, 35, 37, 42, 44, 48, 51–63

raceq Werkzeug für die Data Race Analyse. Wird nach thread_tool aufgerufen und erzeugt eine lesbare Ausgabe mit potentiellen Data Race Konflikten. XIII, 12, 52, 54, 61, 62

SKilL-Version Bauhaus-Version, die *SKilL* als Zwischensprache verwendet. Diese Masterarbeit migriert Programmelemente von der IML-Version in die *SKilL*-Version. XII, 7, 11, 14, 20, 21, 24, 34, 37, 46, 49, 52–56, 62

specgen Spezifikationsgenerater, der eine IML-Spezifikation einliest und eine äquivalente *SKilL*-Spezifikation generiert. Wurde im Rahmen der Masterarbeit [Przytarski, 2016, Kapitel 4] entwickelt. IX, XI, 11, 12, 35, 36

thread_tool Werkzeug für die Data Race Analyse. Wird nach iml2cfg und vor raceq aufgerufen.. XI, XII, 12, 61, 62

Abkürzungsverzeichnis

AF	Abstrakter Funktionsaufruf
AK	Auskommentiert
API	Advanced Programming Interface
AST	Abstract Syntax Tree
AVL	Binärer Suchbaum benannt nach den Mathematikern Georgi Maximowitsch Adelson-Velski und Jewgeni Michailowitsch Landis
CLOC	Count Lines of Code Tool
gcc	GNU Compiler Collection
IML	Intermediate Language
ISTE	Institut für Softwaretechnologie
LoC	Lines of Code
RFG	Resource-Flow-Graph
SKilL	Serialization Killer Language
TCA	Toter-Code-Analyse
TC	Toter-Code

Literatur

[Barnes, 2013] Barnes, J. (2013). *Ada Rationale 2012*. John Barnes Informatics, Ada-Europe, Ada Resource Association.

[Barnes, 2014] Barnes, J. (2014). *Programming in Ada*. Cambridge University Press.

[Felden, 2017] Felden, T. (2017). The SKilL Language V1.0. Technical report.

[Harth, 2014] Harth, F. (2014). Plattform- und sprachungabhängige Serialisierung mit SKilL. Technical report.

[ISO/IEC, 2012] ISO/IEC (2012). Ada Reference Manual ISO/IEC 8652:2012(E) Language and Standard Libraries.

[Keul, 2011] Keul, S. (2011). Tuning Static Data Race Analysis for Automotive Control Software. Technical report, Universität Stuttgart.

[Koschke, 2008] Koschke, R. (2008). Zehn Jahre WSR - Zwölf Jahre Bauhaus. Technical report.

[Przytarski, 2016] Przytarski, D. (2016). Masterarbeit Nr. 108 - SKilLed Bauhaus. Technical report, Universität Stuttgart.

[Raza et al., 2006] Raza, A., Vogel, G., and Plödereder, E. (2006). Bauhaus - a Tool Suite for Program Analysis and Reverse Engineering. Technical report, Universität Stuttgart.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift